

Contents

实体框架

EF Core 和 EF6

比较 EF Core 和 EF6

从 EF6 移植到 EF Core

概述

移植基于 EDMX 的模型

移植基于 Code 的模型

同一应用程序中使用 EF6 和 EF Core

Entity Framework Core

概述

版本和规划(路线图)

当前和计划的版本

版本规划过程

EF Core 5.0

高级计划

新增功能？

EF Core 3.0

新增功能

重大更改

EF Core 2.2

EF Core 2.1

EF Core 2.0

EF Core 1.1

EF Core 1.0

从以前的版本升级

从 1.0 RC1 升级到 RC2

从 1.0 RC2 升级到 RTM

从 1.x 升级到 2.0

入门

[EF Core 教程](#)

[安装 EF Core](#)

[ASP.NET Core 教程 >>](#)

[基本](#)

[连接字符串](#)

[日志记录](#)

[连接复原](#)

[测试](#)

[概述](#)

[使用 SQLite 进行测试](#)

[使用 InMemory 进行测试](#)

[配置 DbContext](#)

[可为空引用类型](#)

[创建模型](#)

[概述](#)

[实体类型](#)

[实体属性](#)

[密钥](#)

[生成的值](#)

[并发标记](#)

[阴影属性](#)

[关系](#)

[索引](#)

[继承](#)

[序列](#)

[支持字段](#)

[值转换](#)

[数据种子设定](#)

[实体类型构造函数](#)

[表拆分](#)

[从属实体类型](#)

[无键实体类型](#)

[具有相同 DbContext 的交替模型](#)

[空间数据](#)

[管理数据库架构](#)

[概述](#)

[迁移](#)

[概述](#)

[团队环境](#)

[自定义操作](#)

[使用独立项目](#)

[多个提供程序](#)

[自定义历史记录表](#)

[创建和删除 API](#)

[反向工程\(基架\)](#)

[查询数据](#)

[概述](#)

[客户端与服务器评估](#)

[跟踪与不跟踪](#)

[复杂查询运算符](#)

[加载相关数据](#)

[异步查询](#)

[原始 SQL 查询](#)

[全局查询筛选器](#)

[查询标记](#)

[查询的工作原理](#)

[保存数据](#)

[概述](#)

[基本保存](#)

[相关数据](#)

[级联删除](#)

[并发冲突](#)

[事务](#)

[异步保存](#)

[断开连接的实体](#)
[生成的属性的显式值](#)
[支持的 .NET 实现](#)
[数据库提供程序](#)

[概述](#)

[Microsoft SQL Server](#)

[概述](#)

[内存优化表](#)

[指定 Azure SQL 数据库选项](#)

[SQLite](#)

[概述](#)

[SQLite 限制](#)

[Cosmos](#)

[概述](#)

[使用非结构化数据](#)

[Cosmos 限制](#)

[InMemory\(用于测试\)](#)

[编写数据库提供程序](#)

[提供程序影响的更改](#)

[工具和扩展](#)

[命令行参考](#)

[概述](#)

[程序包管理器控制台 \(Visual Studio\)](#)

[.NET Core CLI](#)

[设计时 DbContext 创建](#)

[设计时服务](#)

[EF Core API 参考 >>](#)

[Entity Framework 6](#)

[概述](#)

[新增功能](#)

[概述](#)

[以前的版本](#)

[升级到 EF6](#)

[Visual Studio 版本](#)

[入门](#)

[基本](#)

[获取实体框架](#)

[使用 DbContext](#)

[了解关系](#)

[异步查询和保存](#)

[配置](#)

[基于代码](#)

[配置文件](#)

[连接字符串](#)

[依赖项解析](#)

[连接管理](#)

[连接复原](#)

[重试逻辑](#)

[事务提交失败](#)

[数据绑定](#)

[WinForms](#)

[WPF](#)

[断开连接的实体](#)

[概述](#)

[自跟踪实体](#)

[概述](#)

[演练](#)

[日志记录和拦截](#)

[性能](#)

[性能注意事项\(白皮书\)](#)

[使用 NGEN](#)

[使用预生成的视图](#)

[提供程序](#)

[概述](#)

[EF6 提供程序模型](#)

[提供程序中的空间支持](#)

[使用代理](#)

[使用 EF6 进行测试](#)

[使用模拟](#)

[编写自己的测试双精度值](#)

[使用 EF4 的可测试性\(文章\)](#)

[创建模型](#)

[概述](#)

[Code First](#)

[工作流](#)

[使用新数据库](#)

[使用现有数据库](#)

[数据注释](#)

[DbSets](#)

[数据类型](#)

[枚举](#)

[空间](#)

[约定](#)

[内置约定](#)

[自定义约定](#)

[模型约定](#)

[Fluent 配置](#)

[关系](#)

[类型和属性](#)

[在 Visual Basic 中使用](#)

[存储过程映射](#)

[迁移](#)

[概述](#)

[自动迁移](#)

[使用现有数据库](#)

[自定义迁移历史记录](#)

[使用 Migrate.exe](#)

[团队环境中的迁移](#)

[使用 EF 设计器](#)

[工作流](#)

[Model-First](#)

[Database-First](#)

[数据类型](#)

[复杂类型](#)

[枚举](#)

[空间](#)

[拆分映射](#)

[实体拆分](#)

[表拆分](#)

[继承映射](#)

[每个层次结构的表](#)

[每种类型的表](#)

[映射存储过程](#)

[查询](#)

[更新](#)

[映射关系](#)

[多个关系图](#)

[选择运行时版本](#)

[代码生成](#)

[概述](#)

[旧的 ObjectContext](#)

[高级](#)

[EDMX 文件格式](#)

[定义查询](#)

[多个结果集](#)

[表值函数](#)

[键盘快捷方式](#)

[查询数据](#)

[概述](#)

[Load 方法](#)

[本地数据](#)

[跟踪和无跟踪查询](#)

[使用原始 SQL 查询](#)

[查询相关数据](#)

[保存数据](#)

[概述](#)

[更改跟踪](#)

[自动检测更改](#)

[实体状态](#)

[属性值](#)

[处理并发冲突](#)

[使用事务](#)

[数据验证](#)

[其他资源](#)

[博客](#)

[案例研究](#)

[参与](#)

[获取帮助](#)

[术语表](#)

[学校示例数据库](#)

[工具和扩展](#)

[许可证](#)

[EF5](#)

[简体中文](#)

[繁体中文](#)

[德语](#)

[英语](#)

[西班牙语](#)

[法语](#)

[意大利语](#)

[日语](#)

[韩语](#)

[俄语](#)

[EF6](#)

[预发行](#)

[简体中文](#)

[繁体中文](#)

[德语](#)

[英语](#)

[西班牙语](#)

[法语](#)

[意大利语](#)

[日语](#)

[韩语](#)

[俄语](#)

[EF6 API 参考 >>](#)

Entity Framework 文档

Entity Framework

Entity Framework 是一种对象关系映射程序 (O/RM)，可方便 .NET 开发人员使用 .NET 对象处理数据库。开发人员无需再像往常一样编写大部分数据访问代码。



Entity Framework Core

EF Core 是轻量化、可扩展和跨平台版的 Entity Framework。



Entity Framework 6

EF 6 是经过反复测试的数据访问技术，其功能和稳定性已沿用多年。



选择

确定哪个 EF 版本适合你。



移植到 EF Core

关于将现有 EF 6 应用程序移植到 EF Core 的指南。

[EF Core](#)

[全部](#)

EF Core 是轻量化、可扩展和跨平台版的 Entity Framework。



入门

[概述](#)

[创建模型](#)

[查询数据](#)

[保存数据](#)



教程

[更多...](#)



数据库提供程序

[SQL Server](#)

[MySQL](#)

[PostgreSQL](#)

[SQLite](#)

[Cosmos](#)

[更多...](#)

□

[API 参考](#)

[DbContext](#)

[DbSet< TEntity >](#)

[更多...](#)

[EF 6](#)

EF 6 是经过反复测试的数据访问技术，其功能和稳定性已沿用多年。

□

[入门](#)

了解如何使用 Entity Framework 6 访问数据。

□

[API 参考](#)

浏览 Entity Framework 6 API(按命名空间排列)。

比较 EF Core 和 EF6

2020/4/8 • [Edit Online](#)

EF Core

Entity Framework Core ([EF Core](#)) 是适用于 .NET 的新式对象数据库映射器。它支持 LINQ 查询、更改跟踪、更新和架构迁移。

EF Core 通过[数据库提供程序插件模型](#)与 SQL Server/SQL Azure、SQLite、Azure Cosmos DB、MySQL、PostgreSQL 和更多数据库配合使用。

EF6

Entity Framework 6 ([EF6](#)) 是专为 .NET Framework 设计的对象关系映射器，但支持 .NET Core。EF6 是一款受支持的稳定产品，但我们不再对其进行积极开发。

功能比较

EF Core 提供了不会在 EF6 中实现的新功能。但是，并非所有 EF6 功能都已在 EF Core 中实现。

下表比较了 EF Core 和 EF6 中可用的功能。这只是大致比较，没有列出全部功能，也未解释不同 EF 版本中相同功能之间的差异。

EF Core 列指出了功能首次出现的产品版本。

创建模型

功能	EF6.4	EF CORE
基本类映射	是	1.0
带有参数的构造函数		2.1
属性值转换		2.1
没有键的映射类型		2.1
约定	是	1.0
自定义约定	是	1.0(部分; #214)
数据注释	是	1.0
Fluent API	是	1.0
继承: 每个层次结构一个表 (TPH)	是	1.0
继承: 每个类型一个表 (TPT)	是	计划在 5.0 版中推出 (#2266)
继承: 每个具体类一个表 (TPC)	是	5.0 版的延伸目标 (#3170) ⁽¹⁾

EF	EF6.4	EF CORE
阴影状态属性		1.0
备用键		1.0
多对多导航	是	计划在 5.0 版中推出 (#19003)
多对多, 无联接实体	是	积压工作 (#1368)
密钥生成:数据库	是	1.0
密钥生成:客户端		1.0
复杂/已拥有类型	是	2.0
空间数据	是	2.2
模型格式:代码	是	1.0
从数据库更新模型:命令行	是	1.0
从数据库更新模型	部分	积压工作 (#831)
全局查询筛选器		2.0
表拆分	是	2.0
实体拆分	是	5.0 版的延伸目标 (#620) ⁽¹⁾
数据库标量函数映射	差	2.0
字段映射		1.1
可为空引用类型 (C# 8.0)		3.0
模型的图形可视化效果	是	未计划支持 ⁽²⁾
图形模型编辑器	是	未计划支持 ⁽²⁾
模型格式:EDMX (XML)	是	未计划支持 ⁽²⁾
从数据库更新模型:VS 向导	是	未计划支持 ⁽²⁾

查询数据

EF	EF6.4	EF CORE
LINQ 查询	是	1.0
可读内容生成的 SQL	差	1.0

功能	EF 6.4	EF Core
GroupBy 转换	是	2.1
加载关联数据: 预先	是	1.0
加载关联数据: 预先加载派生类型		2.1
加载关联数据: 延迟	是	2.1
加载关联数据: Explicit	是	1.1
原生 SQL 查询: 实体类型	是	1.0
原生 SQL 查询: 无键实体类型	是	2.1
原生 SQL 查询: 使用 LINQ 编写		1.0
显式编译的查询	差	2.0
await foreach (C# 8.0)		3.0
基于文本的查询语言(实体 SQL)	是	未计划支持 ⁽²⁾

保存数据

功能	EF 6.4	EF Core
更改跟踪: 快照	是	1.0
更改跟踪: 通知	是	1.0
更改跟踪: 代理	是	已在 5.0 版中合并 (#10949)
访问跟踪的状态	是	1.0
开放式并发	是	1.0
事务	是	1.0
批处理语句		1.0
存储过程映射	是	积压工作 (#245)
断开连接低级别 API 图形	差	1.0
断开连接端到端图形		1.0(部分; #5536)

其他功能

功能	EF 6.4	EF Core
迁移	是	1.0

功能	EF6.4	EF Core
数据库创建/删除 API	是	1.0
种子数据	是	2.1
连接复原	是	1.1
拦截器	是	3.0
事件	是	3.0(部分; #626)
简单的日志记录 (Database.Log)	是	已在 5.0 版中合并 (#1199)
DbContext 池		2.0

数据库提供程序⁽³⁾

功能	EF6.4	EF Core
SQL Server	是	1.0
MySQL	是	1.0
postgresql	是	1.0
Oracle	是	1.0
SQLite	是	1.0
SQL Server Compact	是	1.0 ⁽⁴⁾
DB2	是	1.0
Firebird	是	2.0
Jet (Microsoft Access)		2.0 ⁽⁴⁾
Azure Cosmos DB		3.0
内存中(用于测试)		1.0

¹ 在给定版本中，不太可能实现延伸目标。但如果一切顺利，我们将尝试加入相关功能。

² EF Core 中不会实现某些 EF6 功能。这些功能依赖于 EF6 的基础实体数据模型 (EDM)，并且/或者是复杂功能，投资回报率相对较低。欢迎提出反馈，但是，尽管 EF Core 支持许多在 EF6 中无法实现的功能，反过来，EF Core 支持 EF6 的所有功能却并不可行。

³ 更新到新的 EF Core 主版本时，第三方实现的 EF Core 数据库提供程序可能延迟。有关详细信息，请参阅[数据库提供程序](#)。

⁴ SQL Server Compact 和 Jet 提供程序仅适用于 .NET Framework(而不适用于 .NET Core)。

受支持的平台

EF Core 3.1 通过使用 .NET Standard 2.0 在 .NET Core 和 .NET Framework 上运行。但 EF Core 5.0 不会在 .NET Framework 上运行。有关更多详细信息, 请参阅[平台](#)。

EF6.4 通过多目标在 .NET Core 和 .NET Framework 上运行。

针对新应用程序的选择指南

除非应用需要[仅在 .NET Framework 上受支持](#)的内容, 否则对于所有新应用程序都在 .NET Core 上使用 EF Core。

针对现有 EF6 应用程序的选择指南

EF Core 不是 EF6 的直接替换项。从 EF6 迁移到 EF Core 可能需要更改应用程序。

将 EF6 应用迁移到 .NET Core 时:

- 如果数据访问代码稳定且不太可能开发或需要新功能, 请继续使用 EF6。
- 如果数据访问代码不断演变, 或应用需要仅在 EF Core 中提供的新功能, 请迁移到 EF Core。
- 迁移到 EF Core 通常也是为了提高性能。但是, 并非所有方案都可提高性能, 因此请先进行分析。

有关详细信息, 请参阅[从 EF6 到 EF Core 的迁移](#)。

从 EF6 移植到 EF Core

2020/4/8 ·

由于 EF Core 有一些根本性变化，我们不建议尝试将 EF6 应用程序迁移至 EF Core，除非你有令人信服的理由进行此项更改。你应该将从 EF6 移至 EF Core 视作移植而不是升级。

IMPORTANT

启动移植过程前，务必验证 EF Core 符合应用程序的数据访问要求。

缺少的功能

请确保 EF Core 包含需要在应用程序中使用的所有功能。有关 EF Core 与 EF6 中功能集的详细比较，请参阅[功能比较](#)。若缺少所需的任何功能，请确保在移植到 EF Core 前弥补这些功能的缺失。

行为变更

下表包含 EF6 和 EF Core 之间的一些行为变更，非完整列表。移植应用程序时，务必牢记它们，因为这些可能会更改应用程序的行为方式，而在交换到 EF Core 后它们不会显示为编译错误。

DbSet.Add/Attach 和图形行为

在 EF6 中，在实体上调用 `DbSet.Add()` 将递归搜索导航属性引用的所有实体。所找到的上下文未在跟踪的任何实体也会被标记为“已添加”。`DbSet.Attach()` 的行为相同，但会将所有实体标记为“未更改”。

EF Core 执行相似的递归搜索，但有些规则略有不同。

- 根实体始终处于请求的状态（对于 `DbSet.Add` 为“已添加”，对于 `DbSet.Attach` 为“未更改”）。
- 适用于递归搜索导航属性期间找到的所有实体：
 - 若实体的主键由存储生成
 - 若主键未设置为值，状态将设置为“已添加”。若为主键值的属性类型分配了 CLR 默认值（如为 `int` 分配 `0`，为 `string` 分配 `null` 等），则视为“未设置”它。
 - 若主键设置为值，状态将设置为“未更改”。
 - 若主键非由数据库生成，则将实体置于与根相同的状态。

Code First 数据库初始化

EF6 包含许多关于执行数据库连接和数据库初始化的功能。这些规则包括：

- 若未执行任何配置，EF6 将从 SQL Express 或 LocalDb 选择数据库。
- 若应用程序 `App/Web.config` 文件包含与上下文名称相同的连接字符串，将使用此连接。
- 如果数据库不存在，则会创建一个。
- 若数据库中不存在模型中的任何表，则会将当前模型的架构添加到数据库。若启用了迁移，则会使用它们创建数据库。
- 若数据库存在，并且 EF6 此前已创建架构，则将检查架构是否与当前模型兼容。若在创建架构后模型已更改，将引发异常。

EF Core 不执行任何此类功能。

- 必须在代码中显式配置数据库连接。
- 不执行初始化。必须使用 `DbContext.Database.Migrate()` 应用迁移(或使用 `DbContext.Database.EnsureCreated() / EnsureDeleted()` 创建/删除数据库, 而不使用迁移)。

Code First 表命名约定

EF6 通过复数形式服务运行实体类名, 来评估实体映射到的默认表名称。

EF Core 使用派生的上下文上公开实体的 `DbSet` 属性的名称。若实体不包含 `DbSet` 属性, 则使用类名。

将基于 EF6 EDMX 的模型移植到 EF Core

2020/4/8 •

EF Core 不支持对模型使用 EDMX 文件格式。要移植这些模型，最佳方法是从应用程序的数据库中生成基于代码的新模型。

安装 EF Core NuGet 包

安装 `Microsoft.EntityFrameworkCore.Tools` NuGet 包。

重新生成模型

现可使用反向工程功能基于现有数据库创建模型。

在包管理器控制台（“工具”->“NuGet 包管理器”->“包管理器控制台”）中运行以下命令。请参阅[包管理器控制台 \(Visual Studio\)](#)，获取用于设置一部分表的基架等的命令选项。

```
Scaffold-DbContext "<connection string>" <database provider name>
```

例如，以下命令用于在 SQL Server LocalDB 实例上根据博客数据库设置模型的基架。

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer
```

删除 EF6 模型

现从应用程序中删除 EF6 模型。

可将 EF6 NuGet 包 (EntityFramework) 保留在安装状态，因为 EF Core 和 EF6 可在同一应用程序中并排使用。但是，如果你不打算在应用程序的任何区域使用 EF6，则卸载此包将有助于就需注意的代码片段生成编译错误。

更新代码

此时，需要处理编译错误并检查代码，以查看 EF6 与 EF Core 之间的行为变化是否会对你产生影响。

测试移植

你的应用程序进行了编译，并不意味着它会成功移植到 EF Core。需要测试应用程序的所有区域，确保行为变化都未对你的应用程序产生负面影响。

将基于 EF6 代码的模型移植到 EF Core

2020/4/8 •

如果你已阅读所有说明且已准备好进行移植，请查看下面一些指南，它们可帮你入门。

安装 EF Core NuGet 包

要使用 EF Core，请针对要使用的数据库提供程序安装 NuGet 包。例如，如果以 SQL Server 为目标，则安装 `Microsoft.EntityFrameworkCore.SqlServer`。有关详细信息，请参阅[数据库提供程序](#)。

如果计划使用迁移，则还应安装 `Microsoft.EntityFrameworkCore.Tools` 包。

可将 EF6 NuGet 包 (`EntityFramework`) 保留在安装状态，因为 EF Core 和 EF6 可在同一应用程序中并排使用。但是，如果你不打算在应用程序的任何区域使用 EF6，则卸载此包将有助于就需注意的代码片段生成编译错误。

交换命名空间

在 EF6 中使用的大多数 API 位于 `System.Data.Entity` 命名空间(及相关的子命名空间)中。第一个代码更改是交换到 `Microsoft.EntityFrameworkCore` 命名空间。通常，你首先是处理已派生的上下文代码文件，然后从此处开始，在出现编译错误时处理这些错误。

上下文配置(连接等)

如[确保 EF Core 适用于你的应用程序](#)中所述，EF Core 检测到连接到的数据库时操作更一目了然。你将需要替代已派生的上下文中的 `OnConfiguring` 方法，并使用数据库提供程序特定的 API 设置到数据库的连接。

大多数 EF6 应用程序将连接字符串存储在应用程序 `App/Web.config` 文件中。在 EF Core 中，你将使用 `ConfigurationManager` API 读取此连接字符串。你可能还需要添加对 `System.Configuration` 框架程序集的引用才能使用此 API。

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(ConfigurationManager.ConnectionStrings["BloggingDatabase"].ConnectionString);
    }
}
```

更新代码

此时，需要处理编译错误并检查代码，以查看行为变化是否对你产生影响。

现有迁移

目前没有一种可行的方法可将现有 EF6 迁移移植到 EF Core。

如果可能，最好假设之前来自 EF6 的所有迁移都已应用到数据库，然后再使用 EF Core 从此处开始迁移架构。为此，一旦模型移植到 EF Core，即可使用 `Add-Migration` 添加迁移。然后，需已设置基架的迁移的 `Up` 和 `Down` 方法中删除所有代码。设置初始迁移的基架后，后续迁移将与此模型进行比较。

测试移植

你的应用程序进行了编译，并不意味着它会成功移植到 EF Core。需要测试应用程序的所有区域，确保行为变化都未对你的应用程序产生负面影响。

在同一个应用程序中使用 EF Core 和 EF6

2020/4/8 • [Edit Online](#)

通过安装这两个 NuGet 包，可在同一应用程序或库中使用 EF Core 和 EF6。

某些类型在 EF Core 和 EF6 中具有相同的名称，并且仅命名空间有所不同，这可能会使在同一代码文件中同时使用 EF Core 和 EF6 变得复杂。可通过命名空间别名指令轻松消除多义性。例如：

```
using Microsoft.EntityFrameworkCore; // use DbContext for EF Core
using EF6 = System.Data.Entity; // use EF6.DbContext for the EF6 version
```

如果要迁移具有多个 EF 模型的现有应用程序，则可以将其中一些选择性地迁移到 EF Core，其余程序则继续使用 EF6。

Entity Framework Core

2020/4/8 • [Edit Online](#)

Entity Framework (EF) Core 是轻量化、可扩展、[开源](#)和跨平台版的常用 Entity Framework 数据访问技术。

EF Core 可用作对象关系映射程序 (O/RM)，以便于 .NET 开发人员能够使用 .NET 对象来处理数据库，这样就不必经常编写大部分数据访问代码了。

EF Core 支持多个数据库引擎，请参阅[数据库提供程序](#)了解详细信息。

模型

对于 EF Core，使用模型执行数据访问。模型由实体类和表示数据库会话的上下文对象构成，可便于用户查询和保存数据。有关详细信息，请参阅[创建模型](#)。

可以根据现有数据库生成模型，手动将模型编码为与数据库匹配，也可以使用 [EF 迁移](#)根据模型创建数据库，然后在模型随时间推移发生更改时改进它。

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                @"Server=(localdb)\mssqllocaldb;Database=Blogging;Integrated Security=True");
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }
        public int Rating { get; set; }
        public List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}
```

查询

使用语言集成查询 (LINQ) 从数据库检索实体类的实例。有关详细信息，请参阅[查询数据](#)。

```
using (var db = new BloggingContext())
{
    var blogs = db.Blogs
        .Where(b => b.Rating > 3)
        .OrderBy(b => b.Url)
        .ToList();
}
```

保存数据

使用实体类的实例在数据库中创建、删除和修改数据。有关详细信息，请参阅[保存数据](#)。

```
using (var db = new BloggingContext())
{
    var blog = new Blog { Url = "http://sample.com" };
    db.Blogs.Add(blog);
    db.SaveChanges();
}
```

后续步骤

有关介绍性教程，请参阅[Entity Framework Core 入门](#)。

EF Core 版本和计划

2020/4/9 • [Edit Online](#)

稳定版本

RELEASE	FRAMEWORK	RELEASE DATE	ANNOUNCEMENT
EF Core 3.1	.NET Standard 2.0	2022 年 12 月 3 日 (LTS)	公告
EF Core 3.0	.NET Standard 2.1	2020 年 3 月 3 日过期	公告 / 重大更改
EF Core 2.2	.NET Standard 2.0	过期时间: 2019 年 12 月 23 日	公告
EF Core 2.1	.NET Standard 2.0	2021 年 8 月 21 日 (LTS)	公告
EF Core 2.0	.NET Standard 2.0	过期时间: 2018 年 10 月 1 日	公告
EF Core 1.1	.NET Standard 1.3	过期时间: 2019 年 6 月 27 日	公告
EF Core 1.0	.NET Standard 1.3	过期时间: 2019 年 6 月 27 日	公告

有关每个 EF Core 版本支持的特定平台的信息, 请参阅[支持的平台](#)。

有关支持到期时间和长期支持 (LTS) 版本的信息, 请参阅[.NET 支持策略](#)。

更新到新版本的指南

- 修补了受支持版本的安全 bug 和其他严重 bug。始终使用给定版本的最新修补程序。例如, 对于 EF Core 2.1, 使用 2.1.14。
- 主版本更新(例如, 从 EF Core 2 更新到 EF Core 3)通常包含重大更改。在进行主版本更新时, 建议进行全面测试。使用上面的“重大更改”链接获取有关处理重大更改的指导。
- 次要版本更新通常不包含重大更改。但仍建议进行全面测试, 因为新功能可能会引入回归。

版本规划和安排

EF Core 版本与[.NET Core 发货计划](#)保持一致。

修补程序版本通常每月交付, 但提前期较长。我们正在努力对此进行改进。

要详细了解我们如何确定要在每个版本中提供的内容, 请参阅[版本规划过程](#)。我们通常不会对下一个主版本或次要版本之后的版本进行详规划。

EF Core 5.0

下一个计划的稳定版本是 EF Core 5.0, 计划于 2020 年 11 月发布。

已根据记录的[发布计划流程](#)创建了[EF Core 5.0 的高级计划](#)。

你对计划的反馈非常重要。指示问题重要性的最佳方式是在 GitHub 上为该问题投票(竖起大拇指 )。然后，此数据将进入下一个版本的计划过程。

立即获取！

EF Core 5.0 包现在以

- [每日生成](#)的形式提供
 - 所有最新功能和 bug 修复。通常非常稳定；针对每个生成已运行 57,000 多次测试。
- [用于 NuGet 的预览版](#)
 - 滞后于每日生成，但经过测试可用于相应的 ASP.NET Core 和 .NET Core 预览版。

使用预览版或每日生成是一种查找问题并尽快提供反馈的绝佳方式。我们获取此类反馈的速度越快，就越有可能在发布下一个正式版本前实施该反馈。

版本规划过程

2020/4/8 • [Edit Online](#)

我们常常会被问到如何选择将添加到特定版本的特定功能。该文档概述了我们采用的过程。随着我们找到更好的规划方法，该过程在不断演变，但总体思路仍保持不变。

不同类型的版本

不同类型的版本包含不同类型的更改。这反过来意味着对不同类型的版本而言，版本规划是不同的。

修补程序版本

修补程序版本只更改了版本的“修补程序”部分。例如，EF Core 3.1.1 是修补在 EF Core 3.1.0 中发现的问题的版本。

修补程序版本旨在修复关键客户 bug。这意味着修补程序版本中没有新功能。在特殊情况下，修补程序版本中不允许包含 API 更改。

在修补程序版本中进行更改的门槛很高。这是因为有必要保证修补程序版本不会引入新的 bug。因此，决策过程着重于高价值和低风险。

在下述情况中，我们修补问题的可能性更大：

- 该问题会影响多个客户
- 该问题与上一版本相比属于性能退化
- 失败会导致数据损坏

在下述情况中，我们修补问题的可能性更小：

- 有合理的解决方法
- 此修补程序极可能损坏其他某些功能
- bug 处于极端情况

在 [长期支持 \(LTS\)](#) 版本的整个生存期内，这一门槛逐渐升高。这是因为 LTS 版本着重于稳定性。

关于是否修复某问题的最终决定由 Microsoft 的 .NET 主管作出。

次要版本

次要版本只更改了版本的“次要”部分。例如，EF Core 3.1.0 是对 EF Core 3.0.0 进行改进的一个版本。

次要版本：

- 旨在提升上一版本的质量和功能
- 通常包含 bug 修复和新功能
- 不包含有意的中断性变更
- 有一些推送给 NuGet 的预发行预览版

主要版本

主要版本更改的是 EF“主要”版本号。例如，EF Core 3.0.0 是与 EF Core 2.2.x 相比性能大幅提升的一个主要版本。

主要版本：

- 旨在提升上一版本的质量和功能
- 通常包含 bug 修复和新功能

- 某些新功能可能从根本上更改了 EF Core 工作的方式
- 通常包含有意的中断性变更
 - 就我们知道的而言，中断性变更是 EF Core 演变的必要部分
 - 但是，由于可能对客户造成影响，我们在进行任何中断性变更方面考虑谨慎。过程，我们在中断性变更方面过于激进。而今后，我们将努力最大程度减少会使应用程序中断的更改，同时努力减少会使数据库提供程序和扩展中断的更改。
- 有很多推送给 NuGet 的预发行预览版

主要/次要版本规划

GitHub 问题跟踪

对于所有 EF Core 规划来说，GitHub (<https://github.com/dotnet/efcore>) 都是可靠来源。

GitHub 上的问题具有：

- 状态
 - **Open** 问题尚未得到解决。
 - **Closed** 问题已得到解决。
 - 所有已解决的问题都**标记了 closed-fixed**。标记有 closed-fixed 的问题已得到修复且已合并，但可能尚未发布。
 - 其他 **closed-** 标签指出了关闭问题的其他原因。例如，重复问题标记有 closed-duplicate。
- 类型
 - **Bug** 表示错误。
 - **增强功能** 表示新功能或现有功能中的更优功能。
- 里程碑
 - **没有里程碑的问题** 由团队进行处理。尚未作出有关如何处理问题的决定，或者对决定的某项更改正在探讨中。
 - **积压工作 (backlog) 里程碑中的问题** 是指 EF 团队将考虑在某个未来版本中处理的项目
 - 积压工作 (backlog) 中的问题可能**标记有 consider-for-next-release**，这是指该工作项在下一版本列表中的排名靠前。
 - 经版本控制的里程碑中的待解决问题是指团队计划在该版本中处理的项目。例如，**这些是我们计划针对 EF Core 5.0 进行处理的问题**。
 - 经版本控制的里程碑中的已关闭问题是指出该版本而言已完成的问题。请注意，该版本可能尚未发布。例如，**这些是我们计划针对 EF Core 3.0 已完成的问题**。
- 请投票！
 - 投票是指出某个问题对你而言很重要的最佳方式。
 - 只需向问题添加“大拇指朝上” 即可进行投票。例如，**这些是得票数靠前的问题**
 - 如果你认为添加评论会提高重要性，还请添加描述你为何需要该功能的特定原因。评论“+1”(即赞同)或给出类似评论不会提高重要性。

规划过程

与从积压工作 (backlog) 中提取被请求最多次的请求而言，规划过程的参与度更高。这是因为我们会用多种方式从多名利益干系人那里收集反馈。然后，我们会根据下列内容调整版本：

- 客户输入的内容
- 其他利益干系人输入的内容
- 战略方向
- 可用资源
- 计划

我们提出的一些问题是：

1. 我们认为有多少开发人员会使用该功能？该功能会使他们的应用程序/体验有多大的改善？为了回答这个问题，我们收集众多来源(其中包括对问题的评论和投票)的反馈。另一方面是与重要客户的具体互动。
2. 在尚未实现此功能的情况下，用户可用的变通方法是什么？例如，许多开发人员可以映射联接表，以解决缺少本机多对多支持的问题。显然，并非所有开发人员都希望这样做，但很多开发人员都可以这样做，而这也作为决策的一个考虑因素。
3. 实现此功能是否有助于 EF Core 的体系结构发展，从而帮助我们实现其他功能？我们往往偏爱充当其他功能的构建基块的功能。例如，属性包实体可有助于提供多对多支持，而且实体构造函数也启用了延迟加载支持。
4. 功能是可扩展性点吗？我们往往偏爱扩展点(而不是常规功能)，因为它们能让开发人员挂钩自己的行为，并补偿缺少的任何功能。
5. 该功能与其他产品结合使用时的增效作用如何？我们往往偏爱能够实现或显著改善结合使用 EF Core 与其他产品(如 .NET Core、最新版 Visual Studio、Microsoft Azure 等)的体验的功能。
6. 从事功能开发工作的人员的技能如何？如何能最充分地利用这些资源？EF 团队的每个成员以及我们的社区参与者都拥有各个领域的不同程度经验，我们需要相应地进行计划。即便我们希望“全员就位”开发特定功能(如 GroupBy 转换或多对多支持)，这也是不切实际的。

针对 Entity Framework Core 5.0 的计划

2020/4/8 • [Edit Online](#)

如[计划过程中](#)所述，我们已来自利益干系人的输入收集到针对 EF Core 5.0 版的暂定计划中。

IMPORTANT

此计划仍是半成品。这里不进行任何承诺。此计划是一个起点，会随着我们了解更多信息而发展。当前未针对 5.0 进行计划的某些内容可能会被纳入。当前已针对 5.0 进行计划的某些内容可能会被淘汰。

版本号和发布日期。

EF Core 5.0 当前计划[与 .NET 5.0 同时发布](#)。已选择版本“5.0”，以便与 .NET 5.0 保持一致。

受支持的平台

EF Core 5.0 计划可在任何 .NET 5.0 平台上运行(基于[这些平台与 .NET Core 的融合](#))。就 .NET Standard 和使用的实际 TFM 而言，这意味着仍然是 TBD。

EF Core 5.0 不会在 .NET Framework 上运行。

重大更改

EF Core 5.0 将包含一些重大更改，但与 EF Core 3.0 的情况相比，这些更改的严重性要小得多。我们的目标是允许大多数应用程序进行更新而不会中断。

预计会对数据库提供程序进行一些重大更改，尤其是在 TPT 支持方面。但是，我们预计为 5.0 更新提供程序的工作会少于为 3.0 更新所需的工作。

主题

我们提取了几个主要领域或主题，它们将构成对 EF Core 5.0 进行大量投入的基础。

多对多导航属性(即“跳过导航”)

开发人员负责人:@smitpatel 和 @AndriySvyryd

通过 [#19003](#) 进行跟踪

T 恤大小:L

状态:正在进行

多对多是 GitHub 积压工作 (backlog) 中[请求最多的功能](#)(大约 407 张投票)。

对所有多对多关系的支持通过 [#10508](#) 进行跟踪。这可以划分为三个主要区域：

- 跳过导航属性。这些属性使模型可用于查询等，而无需引用基础联接表实体。[\(#19003\)](#)
- 属性包实体类型。这些类型使标准 CLR 类型(例如 `Dictionary`)可用于实体实例，使得每种实体类型都不需要显式 CLR 类型。(5.0 版的延伸目标:[#9914](#)。)
- Sugar 可用于轻松配置多对多关系。(5.0 版的延伸目标。)

我们认为，对于需要多对多支持的人员而言，最重要的阻止因素是无法在业务逻辑(如查询)中使用“自然”关系，而无需引用联接表。联接表实体类型可能仍然存在，但不应妨碍业务逻辑。这就是我们选择为 5.0 处理跳过导航属性的原因。

此时，多对多的其他部分正在作为 EF Core 5.0 的延伸目标而进行从事。这意味着它们当前不在针对 5.0 的计划中，但是如果一切顺利，我们希望将它们纳入。

每个类型一张表 (TPT) 继承映射

开发人员负责人:@AndriySvyryd

通过 [#2266](#) 进行跟踪

T 恤大小:XL

状态:正在进行

我们要实现 TPT 是因为它是经常请求的功能(大约 254 张投票;第三名)，并且它需要一些低级更改，我们认为这些更改适合于总体 .NET 5 计划的基础性质。我们预计这会形成数据库提供程序的重大更改，但与 3.0 所需的更改相比，这些更改的严重性要小得多。

经过筛选的包含

开发人员负责人:@maumar

通过 [#1833](#) 进行跟踪

T 恤大小:M

状态:正在进行

经过筛选的包含是经常请求的功能(大约 317 张投票;第二名)，其工作量不大，我们认为这会使当前需要模型级筛选器或更复杂查询的许多方案不受阻碍或更加容易。

合理化 ToTable、ToQuery、ToView、FromSql 等。

开发人员负责人:@maumar 和 @smit Patel

通过 [#17270](#) 进行跟踪

T 恤大小:L

状态:正在进行

在以前的版本中，我们在支持原始 SQL、无键类型和相关领域方面取得了进展。但是，在所有内容作为一个整体协同工作的方式上存在差距和不一致。5.0 的目标是修复这些问题，并为定义、迁移和使用不同类型的实体及其关联查询和数据库项目创造良好体验。这也可能涉及到已编译查询 API 的更新。

请注意，此项可能会导致某些应用程序级重大更改，因为我们当前拥有的某些功能过于宽松，以至于它可能很快导致人们陷入困境。我们可能最终会阻止其中一些功能，并改为提供有关应执行的操作的指导。

常规查询增强功能

开发人员负责人:@smit Patel 和 @maumar

通过 [5.0 里程碑中使用 `area-query` 标记的问题](#) 进行跟踪

T 恤大小:XL

状态:正在进行

查询转换代码已针对 EF Core 3.0 进行了广泛重写。因此，查询代码一般处于更可靠的状态。对于 5.0，在支持 TPT 和 skip 导航属性所需的更改范围之外，我们未计划进行重大查询更改。但是，仍然需要大量工作来修复 3.0 全面修改中遗留的一些技术债务。我们还计划修复许多 bug 并实现少量的增强功能，以进一步改进总体查询体验。

迁移和部署体验

开发人员负责人:@bricelam

通过 [#19587](#) 进行跟踪

T 恤大小:L

状态:正在进行

当前,许多开发人员在应用程序启动时迁移其数据库。这十分简单,但不建议这样做,因为:

- 多个线程/进程/服务器可能会并发尝试迁移数据库
- 发生这种情况时,应用程序可能会尝试访问不一致的状态
- 通常,不应为应用程序执行授予修改架构的数据库权限
- 如果出现问题,则难以还原为干净状态

我们希望在这里提供更好的体验,从而可以轻松地在部署时迁移数据库。这应该:

- 可在 Linux、Mac 和 Windows 上正常工作
- 在命令行上具有良好体验
- 支持采用容器的方案
- 适用于常用的实际部署工具/流
- 至少集成到 Visual Studio 中

结果可能是在 EF Core 中进行许多小改进(例如,SQLite 上更好的迁移),并与其它团队一起进行指导和长期协作,以改进不仅限于 EF 的端到端体验。

EF Core 平台体验

开发人员负责人:@roji 和 @bricelam

通过 [#19588](#) 进行跟踪

T 恤大小:L

状态:尚未开始

我们提供了有关在类似传统 MVC 的 Web 应用程序中使用 EF Core 的良好指导。适用于其他平台和应用程序模型的指导缺失或过期。对于 EF Core 5.0, 我们计划调查、改进和记录在以下方面使用 EF Core 的体验:

- Blazor
- Xamarin, 包括使用 AOT/链接器情景
- WinForms/WPF/WinUI, 可能还有其他 UI 框架

这可能是在 EF Core 中进行许多小改进,并与其它团队一起进行指导和长期协作,以改进不仅限于 EF 的端到端体验。

我们计划研究的特定领域有:

- 部署,包括使用 EF 工具(例如用于迁移)的体验
- 应用程序模型(包括 Xamarin 和 Blazor),可能还有其它模型
- SQLite 体验,包括空间体验和表重新生成
- AOT 和链接体验
- 诊断集成,包括性能计数器

性能

开发人员负责人:@roji

通过 [5.0 里程碑中使用 `area-perf` 标记的问题](#) 进行跟踪

T 恤大小:L

状态:正在进行

对于 EF Core, 我们计划改进性能基准套件, 并对运行时进行定向性能改进。此外, 我们计划完成在 3.0 发布周期内进行原型设计的新 ADO.NET 批处理 API。同样在 ADO.NET 层, 我们计划对 Npgsql 提供程序进行其他性能改进。

作为此工作的一部分, 我们还计划根据需要添加 ADO.NET/EF Core 性能计数器和其他诊断。

体系结构/参与者文档

文档管理人员主管:@ajcvickers

通过 [#1920](#) 进行跟踪

T 恤大小:L

状态:正在进行

这里的思路是使人们更容易了解 EF Core 的内部情况。这可能对于使用 EF Core 的任何人都十分有用, 但主要动机是使外部人员更容易:

- 为 EF Core 代码做出贡献
- 创建数据库提供程序
- 构建其他扩展

Microsoft.Data.Sqlite 文档

文档管理人员主管:@bricelam

通过 [#1675](#) 进行跟踪

T 恤大小:M

状态:已完成。新文档在 [Microsoft 文档站点上提供](#)。

EF 团队还拥有 Microsoft.Data.Sqlite ADO.NET 提供程序。我们计划在 5.0 版本中完整记录此提供程序。

常规文档

文档管理人员主管:@ajcvickers

通过 [5.0 里程碑的文档存储库中的问题](#) 进行跟踪

T 恤大小:L

状态:正在进行

我们已在针对 3.0 和 3.1 版本更新文档。我们还在致力于:

- 全面修改入门文档, 使它们更易于理解/更易于遵循
- 重新组织文档, 使内容更易于查找并添加交叉引用
- 向现有文档添加更多详细信息和说明
- 更新示例并添加更多示例

修复 bug

通过 5.0 里程碑中使用 [type-bug](#) 标记的问题进行跟踪

开发人员:@roji、@maumar、@bricelam、@smit Patel、@AndriySvyryd、@ajcvickers

T 恤大小:L

状态:正在进行

撰写本文时, 我们已将 135 个 bug 会审为在 5.0 版本中进行修复(已修复了 62 个), 但与上面的常规查询增强功能部分存在很大的重叠。

在 3.0 版本期间, 传入率(最终成为里程碑工作的问题)大约为每月 23 个问题。并非所有这些问题都需要在 5.0 中进行修复。作为大致估计, 我们计划修复在 5.0 时间范围内修复其他 150 个问题。

小型增强功能

通过 5.0 里程碑中使用 [type-enhancement](#) 标记的问题进行跟踪

开发人员:@roji、@maumar、@bricelam、@smit Patel、@AndriySvyryd、@ajcvickers

T 恤大小:L

状态:正在进行

除了上面概述的较大功能之外, 我们还计划对 5.0 进行许多较小的改进, 以修复“小问题”。请注意, 上面概述的更一般主题也涵盖了其中许多增强功能。

非计划

通过 使用 [consider-for-next-release](#) 标记的问题进行跟踪

这些是当前未针对 5.0 版本 计划的 bug 修复和增强功能, 但我们将根据以上工作的进度将它们视为延伸目标。

此外, 我们始终会在计划时考虑[投票最多的问题](#)。从版本中去除其中任何问题总是很痛苦的, 但是我们确实需要针对所拥有的资源制定切合实际的计划。

反馈

你对计划的反馈非常重要。指示问题重要性的最佳方式是在 GitHub 上为该问题投票(竖起大拇指)。然后, 此数据将进入下一个版本的[计划过程](#)。

EF Core 5.0 中的新增功能

2020/4/13 • [Edit Online](#)

EF Core 5.0 目前正在开发中。此页面将包含每个预览版中引入的令人关注的更改的简要介绍。

此页面不会复制 [EF Core 5.0](#) 的计划。计划中介绍了 EF Core 5.0 的整体主题，其中包括我们在交付最终版本之前打算包含的所有内容。

发布时，我们会将此处的链接添加到官方文档。

预览版 2

使用 C# 特性指定属性支持字段

现在，可以使用 C# 特性指定属性的支持字段。使用此特性，即使在不能自动找到支持字段时，EF Core 也能正常读写支持字段。例如：

```
public class Blog
{
    private string _mainTitle;

    public int Id { get; set; }

    [BackingField(nameof(_mainTitle))]
    public string Title
    {
        get => _mainTitle;
        set => _mainTitle = value;
    }
}
```

文档可通过问题 [#2230](#) 进行跟踪。

完成鉴别器映射

EF Core 使用鉴别器列进行 [继承层次结构的 TPH 映射](#)。只要 EF Core 知道该鉴别器的所有可能的值，就可能实现某些性能改进。EF Core 5.0 现在可实现这些改进。

例如，对于返回层次结构中所有类型的查询，旧版 EF Core 总是会生成以下 SQL：

```
SELECT [a].[Id], [a].[Discriminator], [a].[Name]
FROM [Animal] AS [a]
WHERE [a].[Discriminator] IN (N'Animal', N'Cat', N'Dog', N'Human')
```

配置完整的鉴别器映射后，EF Core 5.0 现在将生成以下内容：

```
SELECT [a].[Id], [a].[Discriminator], [a].[Name]
FROM [Animal] AS [a]
```

从预览版 3 开始，这将成为默认行为。

Microsoft.Data.Sqlite 中的性能改进

我们对 SQLite 进行了两项性能改进：

- 现在，利用 SqliteBlob 和流，可以更高效地使用 GetBytes、GetChars 和 GetTextReader 检索二进制和字符串数

据。

- `SqliteConnection` 的初始化现在很慢。

这些改进已实施到 ADO.NET `Microsoft.Data.Sqlite` 提供程序中，因此还可以在 EF Core 之外提升性能。

预览版 1

简单的日志记录

此特性会添加类似于 EF6 中 `Database.Log` 的功能。也就是说，它提供了一种从 EF Core 获得日志的简单方法，而不需要配置任何类型的外部日志记录框架。

初步文档包含在 [2019 年 12 月 5 日发布的 EF 每周状态](#) 中。

其他文档可通过问题 [#2085](#) 进行跟踪。

获取生成的 SQL 的简单方法

EF Core 5.0 引入了 `ToQueryString` 扩展方法，该方法会返回执行 LINQ 查询时 EF Core 生成的 SQL。

初步文档包含在 [2020 年 1 月 9 日发布的 EF 每周状态](#) 中。

其他文档可通过问题 [#1331](#) 进行跟踪。

使用 C# 特性指示实体没有键

现在可以将实体类型配置为不使用新 `KeylessAttribute` 的键。例如：

```
[Keyless]
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public int Zip { get; set; }
}
```

文档可通过问题 [#2186](#) 进行跟踪。

可在初始化的 `DbContext` 上更改连接或连接字符串

现在可以更轻松地创建 `DbContext` 实例，而无需任何连接或连接字符串。而且，现在可以在上下文实例上更改连接或连接字符串。此功能使得同一个上下文实例能够动态地连接到不同的数据库。

文档可通过问题 [#2075](#) 进行跟踪。

更改跟踪代理

EF Core 现在可以生成自动实现 `INotifyPropertyChanging` 和 `INotifyPropertyChanged` 的运行时代理。这些代理会将实体属性的值更改直接报告给 EF Core，从而无需扫描更改。不过，代理有其自身的一组限制，因此并不适合所有人使用。

文档可通过问题 [#2076](#) 进行跟踪。

增强的调试视图

调试视图是调试问题时查看 EF Core 内部情况的一种简单方法。在一段时间之前，我们就已经实现了模型的调试视图。对于 EF Core 5.0，我们使模型视图更易于读取，并在状态管理器中为跟踪的实体添加了新的调试视图。

初步文档包含在 [2019 年 12 月 12 日发布的 EF 每周状态](#) 中。

其他文档可通过问题 [#2086](#) 进行跟踪。

改进了对数据库 null 语义的处理

关系数据库通常将 `NULL` 视为未知值，因此它不等于任何其他 `NULL` 值。而 C# 将 `null` 视为与其他任何 `null` 相比均

相等的定义值。默认情况下，EF Core 会转换查询，使查询使用 C# null 语义。EF Core 5.0 极大地提升了这些转换操作的效率。

文档可通过问题 [#1612](#) 进行跟踪。

索引器属性

EF Core 5.0 支持 C# 索引器属性的映射。这写属性使得实体能够充当属性包，实体中的列被映射为包中的命名属性。

文档可通过问题 [#2018](#) 进行跟踪。

为枚举映射生成检查约束

EF Core 5.0 迁移现可为枚举属性映射生成检查约束。例如：

```
MyEnumColumn VARCHAR(10) NOT NULL CHECK (MyEnumColumn IN ('Useful', 'Useless', 'Unknown'))
```

文档可通过问题 [#2082](#) 进行跟踪。

IsRelational

除了现有 `IsSqlServer`、`IsSqlite` 和 `IsInMemory` 外，还添加了新的 `IsRelational` 方法。此方法可用于测试 `DbContext` 是否使用任何关系数据库提供程序。例如：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    if (Database.IsRelational())
    {
        // Do relational-specific model configuration.
    }
}
```

文档可通过问题 [#2185](#) 进行跟踪。

使用 ETag 的 Cosmos 开放式并发

Azure Cosmos DB 数据库提供程序现在支持使用 ETag 的开放式并发。使用 `OnModelCreating` 中的模型生成器配置 ETag：

```
builder.Entity<Customer>().Property(c => c.ETag).IsEtagConcurrency();
```

然后，`SaveChanges` 将在并发冲突上引发 `DbUpdateConcurrencyException`，[可以处理](#)它来实现重试等。

文档可通过问题 [#2099](#) 进行跟踪。

查询转换以获取更多 DateTime 构造

现在，包含新 `DateTime` 构造的查询会被转换。

此外，现在映射了以下 SQL Server 函数：

- `DateDiffWeek`
- `DateFromParts`

例如：

```
var count = context.Orders.Count(c => date > EF.Functions.DateFromParts(DateTime.Now.Year, 12, 25));
```

文档可通过问题 [#2079](#) 进行跟踪。

查询转换以获取更多字节数组构造

现在，使用 Contains、Length、SequenceEqual 等对 byte[] 属性进行的查询会转换成 SQL。

初步文档包含在 [2019 年 12 月 5 日发布的 EF 每周状态](#) 中。

更多文档可通过问题 [#2079](#) 进行跟踪。

反向查询转换

现在，使用 `Reverse` 的查询会被转换。例如：

```
context.Employees.OrderBy(e => e.EmployeeID).Reverse()
```

文档可通过问题 [#2079](#) 进行跟踪。

按位运算符的查询转换

现在，使用按位运算符的查询会在更多的情况下被转换，例如：

```
context.Orders.Where(o => ~o.OrderID == negatedId)
```

文档可通过问题 [#2079](#) 进行跟踪。

Cosmos 上的字符串的查询转换

现在，在使用 Azure Cosmos DB 提供程序的情况下，使用字符串方法 Contains、StartsWith 和 EndsWith 的查询将被转换。

文档可通过问题 [#2079](#) 进行跟踪。

Entity Framework Core 3.0 中的新功能

2020/4/8 • [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的查询。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）。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 `IAsyncEnumerable<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.HasKey();
        entity.ToTable("Names");
    });

    modelBuilder.Entity<Things>(entity =>
    {
        entity.HasKey();
        entity.ToTable("Things");
    });
}

```

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 `OrderDetails`，则 EF Core 会将 `null` 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](#)。

EF Core 3.0 中包含的中断性变更

2020/4/8 • [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

变更	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	中型
默认情况下, 现在会立即发生级联删除	中型
单个查询中现在开始预先加载相关实体	中型
DeleteBehavior.Restrict 具有更简洁的语义	中型
从属类型关系的配置 API 已更改	中型
每个属性使用独立的内存中整数键生成	中型
无跟踪查询不再执行标识解析	中型
元数据 API 更改	中型
特定于提供程序的元数据 API 更改	中型
UseRowNumberForPaging 已删除	中型
FromSql 方法在与存储过程配合使用时, 无法进行组合	中型
只能在查询根上指定 FromSql 方法	低
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变为可选项	低
与并发标记列共享表的所有实体均必须将其映射到属性	低
如果没有所有者，则无法使用跟踪查询来查询从属实体	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
AddEntityFramework* 添加具有大小限制的 IMemoryCache	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
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](#)

IMPORTANT

EF Core 3.1 也面向 .NET Standard 2.0。它重新支持 .NET Framework。

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

使用 EF Core 3.1。

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

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 `FromSqlRaw` 上）指定新的 `FromSqlInterpolated` 和 `FromSql` 方法（替换 `DbSet<>`）。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 `FromSql` 之外的任意位置指定 `DbSet` 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 `Category` 返回相同的 `Product` 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 `Category` 返回新的 `Product` 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 `context.Entry(blog).Property(e => e.Id).CurrentValue` 本身尚未设置，`blog.Id` 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，有必要了解在调用 `SaveChanges` 之前会删除哪些实体。

缓解措施

可以通过 `context.ChangeTracker` 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在会在 `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变为可选项

[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 `OrderDetails` 时，始终需要 `Order` 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, 或者它在主键之外没有任何必需属性且所有属性均为 `OrderDetails`, 则 EF Core 会将 `null` 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均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 `Order` 则 EF Core 会将该值传播给 `OrderDetails`。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");
}
```

如果没有所有者, 则无法使用跟踪查询来查询从属实体

跟踪问题 #18876

旧行为

在 EF Core 3.0 之前，可像任何其他导航一样查询从属实体。

```
context.People.Select(p => p.Address);
```

新行为

自版本 3.0 起，如果跟踪查询在没有所有者的情况下投射一个从属实体，EF Core 将引发异常。

为什么

如果没有所有者，则无法操作从属实体，因此在绝大多数情况下，不该使用此方式查询它们。

缓解措施

如果应跟踪从属实体，以便之后以任何方式进行修改，则应在查询中包含所有者。

否则，请添加一个 `AsNoTracking()` 调用：

```
context.People.Select(p => p.Address).AsNoTracking();
```

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` 的操作也会通过调用 `AddLogging` 和 `AddMemoryCache` 向 DI 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。

AddEntityFramework* 添加具有大小限制的 IMemoryCache

跟踪问题 #12905

旧行为

在 EF Core 3.0 之前，调用 `AddEntityFramework*` 方法还将向 DI 注册内存缓存服务，且没有大小限制。

新行为

自 EF Core 3.0 起，`AddEntityFramework*` 注册的 `IMemoryCache` 服务具有大小限制。如果随后添加的任何其他服务依赖于 `IMemoryCache`，则它们很快就会达到默认限制，从而导致异常或性能下降。

为什么

如果查询缓存逻辑中存在 bug 或者查询是动态生成的，则无限制地使用 `IMemoryCache` 可能会导致内存使用量不受控制。设定默认限制可减少潜在的 DoS 攻击。

缓解措施

在大多数情况下，如果同时调用了 `AddEntityFramework*` 或 `AddDbContext`，则无需调用 `AddDbContextPool`。因此，最好的缓解措施是删除 `AddEntityFramework*` 调用。

如果应用程序需要这些服务，则使用 `AddMemoryCache` 预先向 DI 容器显式注册 `IMemoryCache` 实现。

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 `ChangeTracker.DetectChanges()` 之前显式调用 `Entry` 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程序是一个错误

[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程序的应用程序，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程序。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 `Samurai` 导航属性（可能是私有属性）将 `Entrance` 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()`)通常需要通过对返回的 `ValueTask<T>` 调用 `Task<T>` 来将其转换为 `AsTask()`。请注意, 这会抵消此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`。改用 `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 `DbContextOptionsExtensionInfo` 属性返回的新 `IDbContextOptionsExtension.Info` 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

这样不仅可以让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，则可更新项目中的 PackageReference 项元数据。

```
<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>
```

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。必须在需要此包中的类型的任何项目下添加此类显式引用。

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));
```

EF Core 2.2 中的新增功能

2020/4/8 • [Edit Online](#)

空间数据支持

空间数据可用于表示对象的物理位置和形状。许多数据库都可以本机存储、索引和查询空间数据。常见方案包括，查询给定距离内的对象，以及测试多边形是否包含给定位置。EF Core 2.2 现在支持使用 [NetTopologySuite \(NTS\)](#) 库中的类型，处理各种数据库中的空间数据。

空间数据支持是作为一系列提供程序专用扩展包进行实现。每个包都为 NTS 类型和方法以及数据库中相应的空间类型和函数提供映射。此类提供程序扩展现在可用于 [SQL Server](#)、[SQLite](#) 和 [PostgreSQL](#)（来自 [Npgsql 项目](#)）。空间类型可以直接与 [EF Core 内存中提供程序](#)一起使用，无需使用额外扩展。

安装提供程序扩展后，便能向实体添加受支持类型的属性。例如：

```
using NetTopologySuite.Geometries;

namespace MyApp
{
    public class Friend
    {
        [Key]
        public string Name { get; set; }

        [Required]
        public Point Location { get; set; }
    }
}
```

然后，可以暂留包含空间数据的实体：

```
using (var context = new MyDbContext())
{
    context.Add(
        new Friend
        {
            Name = "Bill",
            Location = new Point(-122.34877, 47.6233355) {SRID = 4326 }
        });
    context.SaveChanges();
}
```

还可以根据空间数据和操作执行数据库查询：

```
var nearestFriends =
    (from f in context.Friends
    orderby f.Location.Distance(myLocation) descending
    select f).Take(5).ToList();
```

若要详细了解此功能，请参阅[空间类型文档](#)。

从属实体集合

EF Core 2.0 增加了在一对关联中对所有权进行建模的功能。EF Core 2.2 将此功能扩展为，将所有权表达为一对

多关联。所有权有助于约束实体的使用方式。

例如，从属实体：

- 只能出现在其他实体类型的导航属性中。
- 自动加载，并且只能被 DbContext 和所有者跟踪。

在关系数据库中，从属集合映射到所有者的各个表，就像是常规的一对多关联一样。不过，在面向文档的数据库中，我们计划将(从属集合或引用中的)从属实体与所有者嵌套在同一个文档中。

若要使用此功能，可以调用新增的 OwnsMany() API：

```
modelBuilder.Entity<Customer>().OwnsMany(c => c.Addresses);
```

有关详细信息，请参阅[更新后的从属实体文档](#)。

查询标记

此功能简化了将代码中的 LINQ 查询与日志中捕获的已生成 SQL 查询相关联的过程。

若要利用查询标记，请使用新增的 TagWith() 方法对 LINQ 查询进行批注。使用上一示例中的空间查询：

```
var nearestFriends =
    (from f in context.Friends.TagWith(@"This is my spatial query!")
     orderby f.Location.Distance(myLocation) descending
     select f).Take(5).ToList();
```

此 LINQ 查询将生成以下 SQL 输出：

```
-- This is my spatial query!

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

有关详细信息，请参阅[查询标记文档](#)。

EF Core 2.1 中的新增功能

2020/4/8 • [Edit Online](#)

除了大量缺陷修复以及小规模功能增强和性能增强之外，EF Core 2.1 还新增了一些有吸引力的功能：

延迟加载

EF Core 现包含创作可按需加载导航属性的实体类所必需的构建基块。我们还创建了一个新包 - Microsoft.EntityFrameworkCore.Proxies，它可利用这些构建基块基于最小修改实体类(例如具有虚拟导航属性的类)来生成延迟加载代理类。

阅读[有关延迟加载的部分](#)详细了解本主题。

实体构造函数中的参数

作为延迟加载所必需的一个构建基块，我们启用了实体创建，实体可将参数纳入其构造函数中。可使用参数来注入属性值、延迟加载委托和服务。

阅读[有关含参数的实体构造函数部分](#)详细了解本主题。

值转换

到目前为止，EF Core 只能映射底层数据库提供程序本机支持的属性类型。在列和属性之间来回复制值，无需进行任何转换。自 EF Core 2.1 起，可通过值转换来转换从列获取的值，之后再将其应用到属性，反之亦然。我们具有可以根据约定按需应用的大量转换，以及显式配置 API，后者允许在列和属性之间注册自定义转换。此功能有如下一些用法：

- 将枚举存储为字符串
- 使用 SQL Server 映射无符号整数
- 自动加密和解密属性值

阅读[有关值转换的部分](#)详细了解本主题。

LINQ GroupBy 转换

在 2.1 版之前，EF Core 中的 GroupBy LINQ 运算符将始终在内存中进行计算。在大多数情况下，我们现在支持将其转换为 SQL GROUP BY 子句。

此示例显示了一个用 GroupBy 来计算各种聚合函数的查询：

```
var query = context.Orders
    .GroupBy(o => new { o.CustomerId, o.EmployeeId })
    .Select(g => new
    {
        g.Key.CustomerId,
        g.Key.EmployeeId,
        Sum = g.Sum(o => o.Amount),
        Min = g.Min(o => o.Amount),
        Max = g.Max(o => o.Amount),
        Avg = g.Average(o => o.Amount)
    });

```

相应的 SQL 转化如下所示：

```
SELECT [o].[CustomerId], [o].[EmployeeId],  
    SUM([o].[Amount]), MIN([o].[Amount]), MAX([o].[Amount]), AVG([o].[Amount])  
FROM [Orders] AS [o]  
GROUP BY [o].[CustomerId], [o].[EmployeeId];
```

数据种子设定

新版本可提供初始数据来填充数据库。与 EF6 不同，种子设定数据作为模型配置的一部分与实体类型相关联。随后将数据库升级为新版本模型时，EF Core 迁移会自动计算需要应用的插入、更新或删除操作。

如示例所示，可使用它在 `OnModelCreating` 中为 Post 配置种子数据：

```
modelBuilder.Entity<Post>().HasData(new Post{ Id = 1, Text = "Hello World!" });
```

阅读[有关数据种子设定的部分](#)详细了解本主题。

查询类型

EF Core 模型现可包含查询类型。与实体类型不同，查询类型上未定义键，也不能插入、删除或更新查询类型（即它们为只读），但查询可直接返回查询类型。以下是查询类型的一些用法：

- 映射到没有主键的视图
- 映射到没有主键的表
- 映射到模型中定义的查询
- 用作 `FromSql()` 查询的返回类型

阅读[有关查询类型的部分](#)详细了解本主题。

针对派生类型的 Include

现在可在编写 `Include` 方法的表达式时指定仅在派生类型上定义的导航属性。对于 `Include` 的强类型版本，我们支持使用显式强制转换或 `as` 运算符。我们现在还支持在 `Include` 的字符串版本中引用在派生类型上定义的导航属性的名称：

```
var option1 = context.People.Include(p => ((Student)p).School);  
var option2 = context.People.Include(p => (p as Student).School);  
var option3 = context.People.Include("School");
```

阅读[有关派生类型的 Include 部分](#)详细了解本主题。

System.Transactions 支持

我们增加了对 System.Transactions 功能（如 TransactionScope）的应用。使用支持该功能的数据库提供程序时，这将适用于 .NET Framework 和 .NET Core。

阅读[有关 System.Transactions 的部分](#)详细了解本主题。

初始迁移时生成更好的列顺序

根据客户反馈，我们对迁移进行了更新，使得先以与类中声明的属性相同的顺序为表生成列。请注意，在创建初始表后，添加新成员时，EF Core 不能更改顺序。

相关子查询优化

我们改进了查询转换，避免在许多常见情况下执行“N + 1”SQL 查询，一般情况下，在投影中使用导航属性后，来自根查询的数据会与来自相关子查询的数据相连接。进行优化需要缓冲子查询的结果，且我们要求修改查询，选择新行为。

例如，以下查询通常会转换为一个“客户”查询，加上 N（其中“N”是返回的客户数量）个单独的“订单”查询：

```
var query = context.Customers.Select(  
    c => c.Orders.Where(o => o.Amount > 100).Select(o => o.Amount));
```

将 `ToList()` 放入正确的位置，指示缓冲适用于订单，即可启用优化：

```
var query = context.Customers.Select(  
    c => c.Orders.Where(o => o.Amount > 100).Select(o => o.Amount).ToList());
```

请注意，此查询只会被转换为两个 SQL 查询：一个“客户”查询，一个“订单”查询。

[Owned] 属性

现只需使用 [注释类型，并确保所有者实体添加到了模型中，即可配置固有实体类型 \[Owned\]](#)：

```
[Owned]  
public class StreetAddress  
{  
    public string Street { get; set; }  
    public string City { get; set; }  
}  
  
public class Order  
{  
    public int Id { get; set; }  
    public StreetAddress ShippingAddress { get; set; }  
}
```

.NET Core SDK 中包含的命令行工具 dotnet-ef

`dotnet-ef` 命令现在是 .NET Core SDK 的一部分，因此无须在项目中使用 `DotNetCliToolReference` 即可使用各项迁移，或通过现有数据库搭建 `DbContext` 基架。

有关如何为不同版本的 .NET Core SDK 和 EF Core 启用命令行工具的详细信息，请参阅[安装工具](#)的相关部分。

Microsoft.EntityFrameworkCore.Abstractions 包

可以在项目中使用新包内的一些属性和接口，从而启用 EF Core 功能，而无需依赖 EF Core 整体。例如，`[Owned]` 属性和 `ILazyLoader` 接口位于此处。

状态更改事件

`Tracked` 中新增的 `StateChanged` 和 `ChangeTracker` 事件可用于编写逻辑，以响应进入 `DbContext` 或状态更改的实体。

原始 SQL 参数分析器

EF Core 随附新增一个代码分析器，用于检测原始 SQL API（如 `FromSql` 或 `ExecuteSqlCommand`）的潜在不安全用法。例如，对于下面的查询，将会看到一条警告，因为 `minAge` 未参数化：

```
var sql = $"SELECT * FROM People WHERE Age > {minAge}";  
var query = context.People.FromSql(sql);
```

数据库提供程序兼容性

建议配合使用 EF Core 2.1 以及已更新或至少已经过测试可用于 EF Core 2.1 的提供程序。

TIP

如果新功能出现任何意外的不兼容或问题，或你有任何相关反馈，请使用[我们的问题跟踪器](#)进行报告。

EF Core 2.0 中的新增功能

2020/4/8 • [Edit Online](#)

.NET Standard 2.0

EF Core 现面向 .NET Standard 2.0，这意味着它可用于 .NET Core 2.0、.NET Framework 4.6.1 以及其他实现 .NET Standard 2.0 的库。有关支持功能的更多详细信息，请参阅[支持的 .NET 实现](#)。

建模

表拆分

现可将两个或多个实体类型映射到同一个表，其中主键列处于共享状态，每行对应两个或多个实体。

要使用表拆分，必须在共享该表的所有实体类型之间配置识别关系（其中外键属性构成主键）：

```
modelBuilder.Entity<Product>()
    .HasOne(e => e.Details).WithOne(e => e.Product)
    .HasForeignKey<ProductDetails>(e => e.Id);
modelBuilder.Entity<Product>().ToTable("Products");
modelBuilder.Entity<ProductDetails>().ToTable("Products");
```

阅读[有关表拆分的部分](#)，详细了解此功能。

固有类型

固有实体类型可与另一个固有实体类型共享同一 .NET 类型，但是由于它不能仅由 .NET 类型标识，因此必须从另一个实体类型导航到该类型。包含定义导航的实体是所有者。查询所有者时，固有类型将默认包含在内。

依照约定，将为固有类型创建一个阴影主键，并通过表拆分将其映射到与所有者相同的表。这样就可以通过类似于 EF6 中复杂类型的用法来使用固有类型：

```
modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, cb =>
{
    cb.OwnsOne(c => c.BillingAddress);
    cb.OwnsOne(c => c.ShippingAddress);
});

public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

阅读[有关固有实体类型的部分](#)，详细了解此功能。

模型级别查询筛选器

EF Core 2.0 包含一个称为“模型级别查询筛选器”的新功能。凭借此功能，可在元数据模型（通常为 `OnModelCreating`）的实体类型上直接定义 LINQ 查询谓词（通常传递给 LINQ Where 查询运算符的布尔表达式）。此类筛选器自动应用于涉及这些实体类型（包括通过使用 `Include` 或直接导航属性引用等方式间接引用的实体类型）的所有 LINQ 查询。此功能的一些常见应用如下：

- 软删除 - 实体类型定义 `IsDeleted` 属性。
- 多租户 - 实体类型定义 `TenantId` 属性。

以下简单示例演示了此功能在上述两种方案中的应用：

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    public int TenantId { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>().HasQueryFilter(
            p => !p.IsDeleted
            && p.TenantId == this.TenantId);
    }
}
```

我们为 `Post` 实体类型的实例定义一个实现多租户和软删除的模型级别筛选器。请留意 `DbContext` 实例级别属性的使用：`TenantId`。模型级筛选器将使用正确上下文实例（即执行查询的上下文实例）中的值。

可使用 `IgnoreQueryFilters()` 运算符对各个 LINQ 查询禁用筛选器。

限制

- 不允许导航引用。可根据反馈添加此功能。
- 仅可在层次结构的根实体类型上定义筛选器。

数据库标量函数映射

EF Core 2.0 包含来自 [Paul Middleton](#) 的一个重要贡献功能，该功能支持将数据库标量函数映射到方法存根，使其可用于 LINQ 查询并转换为 SQL。

下面是如何使用该功能的简要说明：

在 `DbContext` 上声明一种静态方法，并使用 `DbFunctionAttribute` 对其批注：

```
public class BloggingContext : DbContext
{
    [DbFunction]
    public static int PostReadCount(int blogId)
    {
        throw new NotImplementedException();
    }
}
```

此类方法会自动注册。注册后，对 LINQ 查询中方法的调用可转换为 SQL 中的函数调用：

```
var query =
    from p in context.Posts
    where BloggingContext.PostReadCount(p.Id) > 5
    select p;
```

需要注意的若干事项：

- 依照约定，方法名称在生成 SQL 时会用作函数(此情况下为用户定义的函数)名称，但是你可以在方法注册期间替代名称和架构。
- 当前仅支持标量函数。
- 必须在数据库中创建映射函数。EF Core 迁移不创建此函数。

Code First 的自包含类型配置

在 EF6 中可以通过从 `EntityTypeConfiguration` 派生来封装特定实体类型的 Code First 配置。我们在 EF Core 2.0 中重新添加了此模式：

```
class CustomerConfiguration : IEntityTypeConfiguration<Customer>
{
    public void Configure(EntityTypeBuilder<Customer> builder)
    {
        builder.HasKey(c => c.AlternateKey);
        builder.Property(c => c.Name).HasMaxLength(200);
    }
}

...
// OnModelCreating
builder.ApplyConfiguration(new CustomerConfiguration());
```

高性能

DbContext 池

在 ASP.NET Core 应用程序中使用 EF Core 的基本模式通常是先将自定义 `DbContext` 类型注册到依赖关系注入系统，然后通过控制器中的构造函数参数获取该类型的实例。这会为各个请求创建一个 `DbContext` 新实例。

在版本 2.0 中，我们引入了一种在依赖关系注入中注册自定义 `DbContext` 类型的新方法，即以透明形式引入可重用 `DbContext` 实例的池。要使用 `DbContext` 池，请在服务注册期间使用 `AddDbContextPool` 而不是 `AddDbContext`：

```
services.AddDbContextPool<BlogginContext>(
    options => options.UseSqlServer(connectionString));
```

如果使用此方法，那么在控制器请求 `DbContext` 实例时，我们会首先检查池中有无可用的实例。请求处理完成后，实例的任何状态都将被重置，并且实例本身会返回池中。

从概念上讲，此方法类似于连接池在 ADO.NET 提供程序中的运行原理，并具有节约 `DbContext` 实例初始化成本的优势。

限制

此新方法对使用 `DbContext` 的 `OnConfiguring()` 方法可执行的操作带来了一些限制。

WARNING

如果要在不能在请求间共享的派生的 `DbContext` 类中保留自己的状态(例如私有字段)，请避免使用 `DbContext` 池。EF Core 仅会重置将 `DbContext` 实例添加到池之前所识别的状态。

显式编译的查询

这是第二个选择加入性能功能，旨在为大规模方案提供优势。

EF 早期版本以及 LINQ to SQL 中已经提供手动或显式编译的查询 API，允许应用程序缓存查询转换，使其可仅被计算一次并执行多次。

尽管 EF Core 通常可基于查询表达式的哈希表示法自动编译和缓存查询，但是使用此机制可绕过哈希计算和缓存查询，允许应用程序通过调用委托来使用已编译查询，从而实现性能小幅提升。

```
// Create an explicitly compiled query
private static Func<CustomerContext, int, Customer> _customerById =
    EF.CompileQuery((CustomerContext db, int id) =>
        db.Customers
            .Include(c => c.Address)
            .Single(c => c.Id == id));

// Use the compiled query by invoking it
using (var db = new CustomerContext())
{
    var customer = _customerById(db, 147);
}
```

更改跟踪

Attach 可跟踪新实体和现有实体的关系图。

EF Core 支持通过多种机制自动生成键值。使用此功能时，如果键属性为 CLR 默认值，则会生成一个值（通常为零或 null）。这意味着实体的关系图可传递到 `DbContext.Attach` 或 `DbSet.Attach`，并且 EF Core 会标记键已设置为 `Unchanged` 的实体，同时将没有键集的实体标记为 `Added`。这样就可以轻松地在使用生成键时，附加混合了新实体和现有实体的关系图。`DbContext.Update` 和 `DbSet.Update` 的工作原理相同，但是有键集的实体会被标记为 `Modified` 而不是 `Unchanged`。

查询

改进的 LINQ 转换

可成功执行的查询更多，在数据库中（而不是内存中）计算的逻辑也更多，并且从数据库中检索的不必要数据更少。

GroupJoin 改进

这改进了为组联接生成的 SQL。组联接通常是可选导航属性上子查询的结果。

FromSql 和 ExecuteSqlCommand 中的字符串内插

C# 6 引入了字符串内插功能，此功能允许将 C# 表达式直接嵌入字符串文本，从而提供了一种很适合在运行时生成字符串的方法。在 EF Core 2.0 中，我们为两个主要 API 添加了对内插字符串的特殊支持，这两个 API 用于接收原始 SQL 字符串：`FromSql` 和 `ExecuteSqlCommand`。这项新支持允许以“安全”方式使用 C# 字符串内插。即，采用此方式可防止在运行时动态构造 SQL 时可能发生的常见 SQL 注入错误。

以下是示例：

```
var city = "London";
var contactTitle = "Sales Representative";

using (var context = CreateContext())
{
    context.Set<Customer>()
        .FromSql($@"
            SELECT *
            FROM ""Customers"""
            WHERE ""City"" = {city} AND
                  ""ContactTitle"" = {contactTitle}")
        .ToArray();
}
```

本示例在 SQL 格式字符串中嵌入了两个变量。EF Core 会生成如下 SQL：

```
@p0='London' (Size = 4000)
@p1='Sales Representative' (Size = 4000)

SELECT *
FROM ""Customers"""
WHERE ""City"" = @p0
AND ""ContactTitle"" = @p1
```

EF.Functions.Like()

我们已添加 EF.Functions 属性，EF Core 或提供程序可使用该属性定义映射到数据库函数或运算符的方法，从而可在 LINQ 查询中的第一个示例是 Like()：

```
var aCustomers =
    from c in context.Customers
    where EF.Functions.Like(c.Name, "a%")
    select c;
```

请注意，Like() 附带一个内存中实现，处理内存中数据库时或需要在客户端上进行谓词计算时，此实现可能非常方便。

数据库管理

DbContext 基架的复数化挂钩

EF Core 2.0 引入了一种新的 IPluralizer 服务，用于单数化实体类型名称和复数化 DbSet 名称。默认实现为 no-op，因此这仅仅是开发人员可轻松插入自己的复数化程序的挂钩。

下面是开发人员挂入自己的复数化程序的示例：

```
public class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
    {
        services.AddSingleton<IPluralizer, MyPluralizer>();
    }
}

public class MyPluralizer : IPluralizer
{
    public string Pluralize(string name)
    {
        return Inflector.Inflector.Pluralize(name) ?? name;
    }

    public string Singularize(string name)
    {
        return Inflector.Inflector.Singularize(name) ?? name;
    }
}
```

其他

将 ADO.NET SQLite 提供程序移动到 SQLitePCL.raw

这为我们提供了一个 Microsoft.Data.Sqlite 中的更强大解决方案，用以在不同平台分发本机 SQLite 二进制文件。

每个模型仅有一个提供程序

显著增强了提供程序与模型的交互方式，并简化了约定、注释和 Fluent API 用于不同提供程序的方法。

EF Core 2.0 现将对所用的每个不同提供程序生成不同的 [IModel](#)。这对应用程序而言通常是透明的。这有助于简化较低级别的元数据 API，从而始终通过调用（[而不是](#)、[.Relational](#) 等）来访问常见关系元数据概念 [.SqlServer` ` .Sqlite](#)。

增强的日志记录和诊断

日志记录（基于 [ILogger](#)）和诊断（基于 [DiagnosticSource](#)）机制现可共享更多代码。

发送给 [ILogger](#) 的消息的事件 ID 在 2.0 中已更改。现在，事件 ID 在 EF Core 代码内具有唯一性。这些消息现在还遵循 MVC 等所用的结构化日志记录的标准模式。

记录器类别也已更改。现提供通过 [DbLoggerCategory](#) 访问的熟知类别集。

[DiagnosticSource](#) 事件现使用与相应 [ILogger](#) 消息相同的事件 ID 名称。

EF Core 1.1 中的新增功能

2020/4/8 • [Edit Online](#)

建模

字段映射

允许配置属性的支持字段。这对于只读属性或具有 Get/Set 方法(而不是属性)的数据很有用。

映射到 SQL Server 内存优化表

你可以指定实体映射到的表是内存优化表。使用 EF Core 创建和维护基于模型的数据库时(使用迁移或 `Database.EnsureCreated()`)，将为这些实体创建内存优化表。

更改跟踪

来自 EF6 的其他更改跟踪 API

如 `Reload`、`GetModifiedProperties`、`GetDatabaseValues` 等。

查询

显式加载

允许在先前从数据库加载的实体上触发导航属性填充。

DbSet.Find

提供一种基于主键值提取实体的简单方法。

其他

连接复原

自动重试失败的数据库命令。当连接到 SQL Azure 时(其中瞬间失败非常普遍)该操作非常有用。

简化的服务替换

替换 EF 使用的内部服务更加简单。

EF Core 1.0 中包含的功能

2020/4/8 • [Edit Online](#)

平台

.NET Framework 4.5.1

包括控制台、WPF、WinForms、ASP.NET 4 等。

.NET Standard 1.3

包括面向 Windows、OSX 和 Linux 上的 .NET Framework 和 .NET Core 的 ASP.NET Core。

建模

基本建模

基于具有常用标量类型 (`int`、`string` 等) 的 get/set 属性的 POCO 实体。

关系和导航属性

可以在模型中根据外键指定一对多和一对零或一对一关系。简单集合或引用类型的导航属性可以与这些关系相关联。

内置约定

这些约定基于实体类的形状构建初始模型。

Fluent API

允许覆盖上下文中的 `OnModelCreating` 方法，以进一步配置按约定发现的模型。

数据注释

可添加到实体类/属性并影响 EF 模型的属性。例如，`[Required]` 将告知 EF 某个属性是必需的。

关系表映射

允许实体映射到表/列。

键值生成

包括客户端生成和数据库生成。

数据库生成的值

允许在插入(默认值)或更新(计算列)时由数据库生成值。

SQL Server 中的序列

允许在模型中定义序列对象。

唯一约束

允许定义备用键以及面向该键的关系的功能。

索引

在模型中定义索引会自动在数据库中引入索引。此外，还支持唯一索引。

阴影状态属性

允许在未在 .NET 类中声明和存储的模型中定义属性，但可以由 EF Core 进行跟踪和更新。不需要在对象中公开这些属性时，通常用于外键属性。

“每个层次结构一张表”继承模式

允许将继承层次结构中的实体使用鉴别器列保存到单个表中，以在数据库中针对给定记录标识实体类型。

模型验证

检测模型中的无效模式并提供有用错误消息。

更改跟踪

快照更改跟踪

通过将当前状态与原始状态的副本(快照)进行比较，可以自动检测实体中的更改。

通知更改跟踪

修改属性值后，允许实体通知更改跟踪器。

访问跟踪的状态

通过 `DbContext.Entry` 和 `DbContext.ChangeTracker`。

附加分离的实体/图

新的 `DbContext.AttachGraph` API 有助于将实体重新附加到上下文，以保存新的/修改的实体。

保存数据

基本保存功能

允许将对实体实例的更改持久保存到数据库。

乐观并发

防止由于从数据库中提取数据而覆盖其他用户所做的更改。

异步 `SaveChanges`

在数据库处理从 `SaveChanges` 发出的命令时，可以释放当前线程以处理其他请求。

数据库事务

表示 `SaveChanges` 始终是原子(意味着它或者完全成功，或者不对数据库进行更改)。还存在事务相关的 API，允许在上下文实例之间共享事务等。

关系：批处理语句

通过将多个 INSERT/UPDATE/DELETE 命令批量放到数据库的单个往返路线中来提供更好的性能。

查询

基本 LINQ 支持

提供使用 LINQ 从数据库检索数据的功能。

混合客户端/服务器评估

使查询能够包含无法在数据库中评估的逻辑，因此，必须在将数据检索到内存后进行评估。

NoTracking

如果上下文无需监视实体实例的变化，可加快查询执行速度(在结果只读的情况下非常有用)。

预先加载

提供 `Include` 和 `ThenInclude` 方法来标识在查询时也应提取的相关数据。

异步查询

当数据库处理查询时，可以释放当前线程(及其相关资源)以处理其他请求。

原始 SQL 查询

提供 `DbSet.FromSql` 方法以使用原始 SQL 查询提取数据。也可以使用 LINQ 编写这些查询。

数据库架构管理

数据库创建/删除 API

多数旨在测试你希望在不使用迁移的情况下快速创建/删除数据库的位置。

关系数据库迁移

允许关系数据库架构随模型更改而演进。

从数据库反向工程

基于现有关系数据库架构搭建 EF 模型基架。

数据库提供程序

SQL Server

连接到 Microsoft SQL Server 2008 及以上版本。

SQLite

连接到 SQLite 3 数据库。

内存中

旨在实现无需连接到真实的数据库即可轻松启用测试。

第三方提供程序

多个提供程序可用于其他数据库引擎。有关完整的列表，请参阅[数据库提供程序](#)。

从 EF Core 1.0 RC1 升级到 1.0 RC2

2020/3/11 ·

本文介绍如何将使用 RC1 包生成的应用程序移动到 RC2。

包名称和版本

在 RC1 和 RC2 之间，我们将从 "实体框架 7" 改为 "Entity Framework Core"。可以通过[Scott Hanselman 详细了解此帖子](#)中的更改原因。由于此更改，我们的包名称将从 `EntityFramework.*` 更改为 `Microsoft.EntityFrameworkCore.*`，版本从 `7.0.0-rc1-final` 到 `1.0.0-rc2-final`（或工具 `1.0.0-preview1-final`）。

需要完全删除 RC1 包，然后安装 RC2 包。下面是一些常用包的映射。

RC1	RC2
<code>EntityFramework.MicrosoftSqlServer 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.SqlServer 1.0.0-rc2-final</code>
<code>EntityFramework.SQLite 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.Sqlite 1.0.0-rc2-final</code>
<code>EntityFramework7.Npgsql 3.1.0-rc1-3</code>	<code>Npgsql. Microsoft.entityframeworkcore. Postgres</code>
<code>EntityFramework.SqlServerCompact35 7.0.0-rc1-final</code>	<code>EntityFrameworkCore.SqlServerCompact35 1.0.0-rc2-final</code>
<code>EntityFramework.SqlServerCompact40 7.0.0-rc1-final</code>	<code>EntityFrameworkCore.SqlServerCompact40 1.0.0-rc2-final</code>
<code>EntityFramework 7.0.0 版-rc1-最终</code>	<code>Microsoft.EntityFrameworkCore.InMemory 1.0.0-rc2-final</code>
<code>EntityFramework.IBMDataServer 7.0.0-beta1</code>	尚不适用于 RC2
<code>EntityFramework.Commands 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.Tools 1.0.0-preview1-final</code>
<code>EntityFramework.MicrosoftSqlServer.Design 7.0.0-rc1-final</code>	<code>Microsoft.EntityFrameworkCore.SqlServer.Design 1.0.0-rc2-final</code>

命名空间

除了包名称，命名空间从 `Microsoft.Data.Entity.*` 更改为 `Microsoft.EntityFrameworkCore.*`。您可以使用 `using Microsoft.EntityFrameworkCore` 的 `using Microsoft.Data.Entity` 的查找/替换来处理此更改。

表命名约定更改

RC2 中的一项重大功能更改是将给定实体的 `DbSet< TEntity >` 属性名称用作其映射到的表名，而不只是类名。有关此更改的详细信息，请参阅[相关公告问题](#)。

对于现有的 RC1 应用程序，我们建议将以下代码添加到 `OnModelCreating` 方法的开头，以保留 RC1 命名策略：

```
foreach (var entity in modelBuilder.Model.GetEntityTypes())
{
    entity.Relational().TableName = entity.DisplayName();
}
```

如果要采用新的命名策略，我们建议成功完成其余的升级步骤，并删除代码并创建迁移，以应用表重命名。

AddDbContext / Startup.cs 更改（仅适用于 ASP.NET Core 项目）

在 RC1 中，你必须将实体框架服务添加到应用程序服务提供程序中 `Startup.ConfigureServices(...)`：

```
services.AddEntityFramework()
    .AddSqlServer()
    .AddDbContext<ApplicationContext>(options =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

在 RC2 中，你可以删除对 `AddEntityFramework()`、`AddSqlServer()` 等的调用：

```
services.AddDbContext<ApplicationContext>(options =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"]));
```

还需要向派生上下文添加一个构造函数，该构造函数采用上下文选项，并将其传递到基构造函数。这是必需的，因为我们消除了在幕后 snuck 它们的一些可怕的神奇之处：

```
public ApplicationContext(DbContextOptions<ApplicationContext> options)
    : base(options)
{}
```

传入 IServiceProvider

如果有将 `IServiceProvider` 传递给上下文的 RC1 代码，则该代码现在已移动到 `DbContextOptions`，而不是单独的构造函数参数。使用 `DbContextOptionsBuilder.UseInternalServiceProvider(...)` 设置服务提供程序。

测试

执行此操作的最常见方案是在测试时控制 InMemory 数据库的范围。有关使用 RC2 执行此操作的示例，请参阅更新的[测试](#)文章。

解析内部服务从应用程序服务提供商（仅适用于 ASP.NET Core 项目）

如果你有一个 ASP.NET Core 的应用程序，并且你希望 EF 解析应用程序服务提供程序中的内部服务，则可以使用 `AddDbContext` 的重载来配置：

```
services.AddEntityFrameworkSqlServer()
    .AddDbContext<ApplicationContext>((serviceProvider, options) =>
    options.UseSqlServer(Configuration["ConnectionStrings:DefaultConnection"])
        .UseInternalServiceProvider(serviceProvider));
```

WARNING

建议允许 EF 内部管理其自己的服务，除非您有理由将内部 EF 服务合并到您的应用程序服务提供程序中。你可能想要执行此操作的主要原因是使用应用程序服务提供程序来替换 EF 在内部使用的服务

DNX 命令 = >.NET CLI（仅适用于 ASP.NET Core 项目）

如果以前使用了 ASP.NET 5 项目的 `dnx ef` 命令，则这些命令现在已移动到 `dotnet ef` 命令。相同的命令语法仍适用。您可以使用 `dotnet ef --help` 来提供语法信息。

命令注册的方式在 RC2 中发生了变化，因为 DNX 被 .NET CLI 替换。命令现在已在 `project.json` 中的 `tools` 部分注册：

```
"tools": {  
  "Microsoft.EntityFrameworkCore.Tools": {  
    "version": "1.0.0-preview1-final",  
    "imports": [  
      "portable-net45+win8+dnxcore50",  
      "portable-net45+win8"  
    ]  
  }  
}
```

TIP

如果你使用 Visual Studio，你现在可以使用程序包管理器控制台运行 ASP.NET Core 项目（这不支持在 RC1 中）的 EF 命令。你仍需要在 `project.json` 的 `tools` 部分注册命令来执行此操作。

程序包管理器命令需要 PowerShell 5

如果你在 Visual Studio 中的包管理器控制台中使用实体框架命令，则需要确保已安装 PowerShell 5。这是将在下一个版本中删除的临时要求（有关详细信息，请参阅[问题 #5327](#)）。

在项目中使用 "imports"

一些 EF Core 依赖关系不支持.NET 标准尚未。标准.NET 和.NET Core 项目中的 EF Core 可能会要求添加“导入”到 `project.json` 临时的解决方法。

添加 EF 时，NuGet 还原将显示以下错误消息：

```
Package Ix-Async 1.2.5 is not compatible with netcoreapp1.0 (.NETCoreApp,Version=v1.0). Package Ix-Async 1.2.5  
supports:  
- net40 (.NETFramework,Version=v4.0)  
- net45 (.NETFramework,Version=v4.5)  
- portable-net45+win8+wp8 (.NETPortable,Version=v0.0,Profile=Profile78)  
Package Remotion.Linq 2.0.2 is not compatible with netcoreapp1.0 (.NETCoreApp,Version=v1.0). Package  
Remotion.Linq 2.0.2 supports:  
- net35 (.NETFramework,Version=v3.5)  
- net40 (.NETFramework,Version=v4.0)  
- net45 (.NETFramework,Version=v4.5)  
- portable-net45+win8+wp8+wpa81 (.NETPortable,Version=v0.0,Profile=Profile259)
```

解决方法是手动导入可移植配置文件 “net451 + win8”。这会强制 NuGet 将与此提供的二进制文件相匹配的二进制文件视为与 .NET Standard 兼容的框架，即使它们不是这样。尽管 “net451 + win8” 与 .NET Standard 的兼容性不是 100%，但其兼容性足以用于从 PCL 转换到 .NET Standard。当 EF 的依赖项最终升级到 .NET Standard 时，可以删除导入。

可以在数组语法中将多个框架添加到“导入”。如果将其他库添加到项目，则可能需要其他导入。

```
{  
  "frameworks": {  
    "netcoreapp1.0": {  
      "imports": ["dnxcore50", "portable-net451+win8"]  
    }  
  }  
}
```

请参阅[问题 #5176](#)。

从 EF Core 1.0 RC2 升级到 RTM

2020/3/11 •

本文提供了有关将使用 RC2 包生成的应用程序移动到 1.0.0 RTM 的指南。

包版本

通常将安装到应用程序中的顶级包的名称在 RC2 和 RTM 之间不会发生更改。

需要将已安装的包升级到 RTM 版本：

- 运行时包(例如 `Microsoft.EntityFrameworkCore.SqlServer`)从 `1.0.0-rc2-final` 更改为 `1.0.0`。
- `Microsoft.EntityFrameworkCore.Tools` 包从 `1.0.0-preview1-final` 改为 `1.0.0-preview2-final`。请注意，工具仍处于预发布版本。

现有迁移可能需要添加 maxLength

在 RC2 中，迁移中的列定义看起来像 `table.Column<string>(nullable: true)` 并且列的长度是在迁移后的代码中存储的某些元数据中查找的。在 RTM 中，此长度现在包含在基架代码

`table.Column<string>(maxLength: 450, nullable: true)` 中。

使用 RTM 之前基架的任何现有迁移都不指定 `maxLength` 参数。这意味着将使用数据库支持的最大长度(`nvarchar(max)` SQL Server)。这对于某些列可能很好，但作为键、外键或索引一部分的列需要更新以包含最大长度。按照约定，450是用于键、外键和索引列的最大长度。如果已在模型中显式配置了长度，则应改为使用该长度。

ASP.NET 标识

此更改会影响使用 ASP.NET Identity 的项目和从 RTM 之前的项目模板创建的项目。项目模板包含用于创建数据库的迁移。必须编辑此迁移以指定以下列的最大 `256` 长度。

- AspNetRoles**
 - 名称
 - `NormalizedNames`
- AspNetUsers**
 - 电子邮件
 - `NormalizedEmail`
 - `NormalizedUserName`
 - `UserName`

如果将初始迁移应用于数据库，则无法进行此更改将导致出现以下异常。

```
System.Data.SqlClient.SqlException (0x80131904): Column 'NormalizedNames' in table 'AspNetRoles' is of a type  
that is invalid for use as a key column in an index.
```

.NET Core: project.json 中删除"导入"

如果面向带有 RC2 的 .NET Core，则需要将 `imports` 添加到项目 json，作为某些 EF Core 的依赖项(不支持 .NET Standard)的临时解决方法。现在可将其删除。

```
{  
  "frameworks": {  
    "netcoreapp1.0": {  
      "imports": [ "dnxcore50", "portable-net451+win8" ]  
    }  
  }  
}
```

NOTE

从版本 1.0 RTM 版, .NET Core SDK 不再支持使用 Visual Studio 2015 `project.json` 或开发 .NET Core 应用程序。我们建议你从 `project.json` 迁移到 `csproj`。如果使用的是 Visual Studio, 建议升级到 [Visual Studio 2017](#)。

UWP: 添加绑定重定向

尝试在通用 Windows 平台 (UWP) 项目上运行 EF 命令会导致以下错误:

```
System.IO.FileLoadException: Could not load file or assembly 'System.IO.FileSystem.Primitives, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a' or one of its dependencies. The located assembly's manifest definition does not match the assembly reference.
```

需要将绑定重定向手动添加到 UWP 项目。在项目根文件夹中创建一个名为 `App.config` 的文件, 并将重定向添加到正确的程序集版本。

```
<configuration>  
  <runtime>  
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
      <dependentAssembly>  
        <assemblyIdentity name="System.IO.FileSystem.Primitives"  
                          publicKeyToken="b03f5f7f11d50a3a"  
                          culture="neutral" />  
        <bindingRedirect oldVersion="4.0.0.0"  
                         newVersion="4.0.1.0"/>  
      </dependentAssembly>  
      <dependentAssembly>  
        <assemblyIdentity name="System.Threading.Overlapped"  
                          publicKeyToken="b03f5f7f11d50a3a"  
                          culture="neutral" />  
        <bindingRedirect oldVersion="4.0.0.0"  
                         newVersion="4.0.1.0"/>  
      </dependentAssembly>  
    </assemblyBinding>  
  </runtime>  
</configuration>
```

从以前版本的应用程序升级到 EF Core 2.0

2020/3/11 •

我们已在 2.0 中利用了显著改进现有 API 和行为的机会。有一些改进可能需要修改现有应用程序代码，但我们认为，对于大多数应用程序而言，这种影响会很低，在大多数情况下，只需重新编译和最少的指导更改即可替换已过时的 API。

更新现有应用程序到 EF Core 2.0 可能还需要：

1. 将应用程序的目标 .NET 实现升级到支持 .NET Standard 2.0 的应用程序。有关更多详细信息，请参阅[支持的 .Net 实现](#)。
2. 标识的提供程序与 EF Core 2.0 兼容的目标数据库。请参阅下面的[EF Core 2.0 需要 2.0 数据库提供商](#)。
3. 升级到 2.0 所有 EF Core 包（运行时和工具）。有关更多详细信息，请参阅[安装 EF Core](#)。
4. 进行任何必要的代码更改，以弥补本文档其余部分所述的重大更改。

ASP.NET Core 现在包括 EF Core

面向 ASP.NET Core 2.0 的应用程序可以使用 EF Core 2.0，而不需要第三方数据库提供程序以外的其他依赖项。但是，面向以前版本的 ASP.NET Core 应用程序需要为了使用 EF Core 2.0 升级到 ASP.NET Core 2.0。有关将 ASP.NET Core 应用程序升级到 2.0 的更多详细信息，请参阅[主题中的 ASP.NET Core 文档](#)。

在 ASP.NET Core 中获取应用程序服务的新方法

已为 2.0 中断 1.x 中使用的 EF Core 的设计时逻辑的方式更新的建议的模式为 ASP.NET Core web 应用程序。在设计时，EF Core 将尝试直接调用 `Startup.ConfigureServices` 以便访问应用程序的服务提供程序。在 ASP.NET Core 2.0 中，将在 `Startup` 类之外初始化配置。使用 EF Core 的应用程序通常通过配置访问其连接字符串，因此 `Startup` 本身就不够了。如果升级 ASP.NET Core 1.x 应用程序，你可能会收到以下错误，使用 EF Core 工具时。

未在 "ApplicationContext" 上找到任何无参数的构造函数。将无参数构造函数添加到 "ApplicationContext"，或在与 "ApplicationContext" 相同的程序集中添加 "IDesignTimeDbContextFactory<ApplicationContext>" 的实现

ASP.NET Core 2.0 的默认模板中已添加新的设计时挂钩。静态 `Program.BuildWebHost` 方法使 EF Core 能够在设计时访问应用程序的服务提供程序。如果要升级 ASP.NET Core 1.x 应用程序，则需要将 `Program` 类更新为类似于以下内容。

```

using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace AspNetCoreDotNetCore2._0App
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}

```

强烈建议在将应用程序更新到2.0时采用这一新模式，这是为了使 Entity Framework Core 迁移等产品功能有效。另一种常见方法是实现 `IDesignTimeDbContextFactory<TContext>`。

已重命名 IDbContextFactory

为了支持不同的应用程序模式并使用户能够更好地控制其 `DbContext` 在设计时的使用方式，我们在过去提供 `IDbContextFactory<TContext>` 接口。在设计时，EF Core 工具将发现项目中此接口的实现，并使用它来创建 `DbContext` 对象。

此接口具有一个非常通用的名称，此名称误导某些用户尝试重复使用它来执行其他 `DbContext` 创建的方案。当 EF 工具在设计时尝试使用其实现并引发 `Update-Database` 或 `dotnet ef database update` 之类的命令失败时，它们被捕获到“保护”。

为了传达此接口的强设计时语义，我们已将其重命名为 `IDesignTimeDbContextFactory<TContext>`。

对于2.0版，`IDbContextFactory<TContext>` 仍存在，但被标记为过时。

DbContextFactoryOptions 已删除

由于上述 ASP.NET Core 2.0 更改，我们发现新 `IDesignTimeDbContextFactory<TContext>` 接口上不再需要 `DbContextFactoryOptions`。下面是你应改用的替代方法。

DBCONTEXTFACTORYOPTIONS	III
ApplicationBasePath	<code>ApplicationContext.BaseDirectory</code>
ContentRootPath	<code>Directory.GetCurrentDirectory()</code>
EnvironmentName	<code>Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")</code>

设计时工作目录已更改

ASP.NET Core 2.0 更改还要求 `dotnet ef` 使用的工作目录与 Visual Studio 在运行应用程序时使用的工作目录一致。这种情况的一个显而易见的副作用是，SQLite 文件名现在是相对于项目目录的，而不是相对于其使用的输出目录。

EF Core 2.0 需要 2.0 版数据库提供程序

为使用 EF Core 2.0 我们所做工作许多简化和方式的数据库提供程序中的改进。这意味着 1.0.x 版和 1.1.x 提供程序不会使用 EF Core 2.0。

SQL Server 和 SQLite 提供程序由 EF 团队提供，2.0 版本将作为 2.0 版本的一部分提供。正在为 2.0 更新面向 SQL Compact、PostgreSQL 和 MySQL 的开源第三方提供程序。对于所有其他提供商，请与提供商联系。

日志记录和诊断事件已更改

注意：这些更改不应影响大部分应用程序代码。

发送到 `ILogger` 的消息的事件 id 在 2.0 中发生了更改。现在，事件 ID 在 EF Core 代码内具有唯一性。这些消息现在还遵循 MVC 等所用的结构化日志记录的标准模式。

记录器类别也已更改。现提供通过 `DbLoggerCategory` 访问的熟知类别集。

`DiagnosticSource` 事件现在使用与对应 `ILogger` 消息相同的事件 ID 名称。事件负载是派生自 `EventData` 的所有名义类型。

事件 Id、负载类型和类别记录在 `CoreEventId` 和 `RelationalEventId` 类中。

Id 还会从 `Microsoft.entityframeworkcore` 移动到新的 `Microsoft.entityframeworkcore` 命名空间。

EF Core 关系元数据 API 更改

EF Core 2.0 现将对所用的每个不同提供程序生成不同的 `IModel`。这对应用程序而言通常是透明的。这样一来，就可以简化较低级别的元数据 API，使对 _公共关系元数据概念_ 的任何访问始终通过调用 `.Relational()` 而不是 `.SqlServer()`、`.Sqlite()` 等。例如，1.1.x 代码如下：

```
var tableName = context.Model.FindEntityType(typeof(User)).SqlServer().TableName;
```

现在应如下所示：

```
var tableName = context.Model.FindEntityType(typeof(User)).Relational().TableName;
```

现在可使用扩展方法（而不是 `ForSqlServerToTable` 方法），根据当前使用的提供程序来编写条件代码。例如：

```
modelBuilder.Entity<User>().ToTable(  
    Database.IsSqlServer() ? "SqlServerName" : "OtherName");
```

请注意，此更改仅适用于 _所有_ 关系提供程序定义的 API/元数据。当 API 和元数据仅特定于单个提供程序时，它保持不变。例如，聚集索引特定于 SQL server，因此仍必须使用 `ForSqlServerIsClustered` 和 `.SqlServer().IsClustered()`。

不控制 EF 服务提供程序

EF Core 使用内部 `IServiceProvider`（依赖关系注入容器）来实现其内部实现。应用程序应允许 EF Core 以创建和管理在特殊情况下此提供程序除外。请考虑删除对 `UseInternalServiceProvider` 的任何调用。如果应用程序需要调用 `UseInternalServiceProvider`，请考虑将 [问题存档](#)，以便我们能够调查处理方案的其他方法。

除非还调用了 `UseInternalServiceProvider`，否则应用程序代码不需要调用 `AddEntityFramework`、`AddEntityFrameworkSqlServer` 等。删除对 `AddEntityFramework` 或 `AddEntityFrameworkSqlServer` 等的任何现有调用，`AddDbContext` 仍以与以前相同的方式使用。

内存中数据库必须命名为

全局未命名内存中数据库已删除，而所有内存中数据库必须命名为。例如：

```
optionsBuilder.UseInMemoryDatabase("MyDatabase");
```

这将创建/使用名为 "MyDatabase" 的数据库。如果使用相同的名称再次调用 `UseInMemoryDatabase`，则将使用相同的内存中数据库，以允许多个上下文实例共享它。

只读 API 更改

`IsReadOnlyBeforeSave`、`IsReadOnlyAfterSave` 和 `IsStoreGeneratedAlways` 已弃用并已替换为 `BeforeSaveBehavior` 和 `AfterSaveBehavior`。这些行为适用于任何属性（不仅是存储生成的属性），并确定在插入到数据库行（`BeforeSaveBehavior`）或更新现有数据库行（`AfterSaveBehavior`）时应如何使用属性的值。

标记为 `ValueGenerated.OnAddOrUpdate`（例如，对于计算列）的属性将默认忽略当前在属性上设置的任何值。这意味着，无论是否已对所跟踪的实体设置或修改任何值，都将始终获取存储生成的值。可以通过设置其他 `Before\AfterSaveBehavior` 来更改此设置。

新 ClientSetNull 删除行为

在以前的版本中，`DeleteBehavior` 具有与上下文跟踪的实体的行为，这些实体与 `SetNull` 语义更的已关闭匹配。在 EF Core 2.0 中，引入了新的 `ClientSetNull` 行为作为可选关系的默认行为。此行为对于使用 EF Core 创建的数据库，具有跟踪实体和 `Restrict` 行为的 `SetNull` 语义。在我们的经验中，这是跟踪的实体和数据库的最预期/有用的行为。为可选关系设置时，`DeleteBehavior.Restrict` 现在适用于跟踪的实体。

已删除提供程序设计时包

`Microsoft.EntityFrameworkCore.Relational.Design` 包已删除。它的内容已合并到 `Microsoft.EntityFrameworkCore.Relational` 和 `Microsoft.EntityFrameworkCore.Design` 中。

这会传播到提供程序的设计时包。将删除这些包（`Microsoft.EntityFrameworkCore.Sqlite.Design`、`Microsoft.EntityFrameworkCore.SqlServer.Design` 等），并将其内容合并到主提供程序包中。

若要在 EF Core 2.0 中启用 `Scaffold-DbContext` 或 `dotnet ef dbcontext scaffold`，只需引用单个提供程序包：

```
<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer"
  Version="2.0.0" />
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
  Version="2.0.0"
  PrivateAssets="All" />
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
  Version="2.0.0" />
```

EF Core 入门

2020/4/8 • [Edit Online](#)

在本教程中，将创建一个 .NET Core 控制台应用，该应用使用 Entity Framework Core 对 SQLite 数据库执行数据访问。

你可在 Windows 上使用 Visual Studio，或在 Windows、macOS 或 Linux 上使用 .NET Core CLI 来学习本教程。

[在 GitHub 上查看此文章的示例。](#)

先决条件

安装以下软件：

- [.NET Core CLI](#)
- [Visual Studio](#)
- [.NET Core SDK](#)。

创建新项目

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet new console -o EFGetStarted  
cd EFGetStarted
```

安装 Entity Framework Core

要安装 EF Core，请为要作为目标对象的 EF Core 数据库提供程序安装程序包。本教程使用 SQLite 的原因是，它可在 .NET Core 支持的所有平台上运行。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

创建模型

定义构成模型的上下文类和实体类。

- [.NET Core CLI](#)
- [Visual Studio](#)
- 在项目文件夹中，使用以下代码创建 Model.cs

```

using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace EFGetStarted
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder options)
            => options.UseSqlite("Data Source=blogging.db");
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }

        public List<Post> Posts { get; } = new List<Post>();
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public Blog Blog { get; set; }
    }
}

```

EF Core 还可以从现有数据库对模型进行[反向工程](#)。

提示:在实际应用中，将每个类放在单独的文件中，并将[连接字符串](#)放在配置文件或环境变量中。为简化本教程，所有内容均放在一个文件中。

创建数据库

以下步骤使用[迁移](#)创建数据库。

- [.NET Core CLI](#)
- [Visual Studio](#)
- 运行以下命令：

```

dotnet tool install --global dotnet-ef
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet ef migrations add InitialCreate
dotnet ef database update

```

这会安装 `dotnet ef` 和设计包，这是对项目运行命令所必需的。`migrations` 命令为迁移搭建基架，以便为模型创建一组初始表。`database update` 命令创建数据库并向其应用新的迁移。

创建、读取、更新和删除

- 打开 `Program.cs` 并将内容替换为以下代码：

```
using System;
using System.Linq;

namespace EFGetStarted
{
    class Program
    {
        static void Main()
        {
            using (var db = new BloggingContext())
            {
                // Create
                Console.WriteLine("Inserting a new blog");
                db.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                db.SaveChanges();

                // Read
                Console.WriteLine("Querying for a blog");
                var blog = db.Blogs
                    .OrderBy(b => b.BlogId)
                    .First();

                // Update
                Console.WriteLine("Updating the blog and adding a post");
                blog.Url = "https://devblogs.microsoft.com/dotnet";
                blog.Posts.Add(
                    new Post
                    {
                        Title = "Hello World",
                        Content = "I wrote an app using EF Core!"
                    });
                db.SaveChanges();

                // Delete
                Console.WriteLine("Delete the blog");
                db.Remove(blog);
                db.SaveChanges();
            }
        }
    }
}
```

运行应用

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet run
```

后续步骤

- 按照 [ASP.NET Core 教程](#)在 Web 应用中使用 EF Core
- 了解有关 [LINQ 查询表达式](#)的详细信息
- 配置模型指定**必需项**和**最大长度**等内容
- 在更改模型后使用[迁移](#)来更新数据库架构

安装 Entity Framework Core

2020/4/8 • [Edit Online](#)

必备条件

- EF Core 是一个 .NET Standard 2.0 库。因此，EF Core 需要支持运行 .NET Standard 2.0 的实现。其他 .NET Standard 2.0 库也可引用 EF Core。
- 例如，可使用 EF Core 开发面向 .NET Core 的应用。生成 .NET Core 应用需要 .NET Core SDK。还可选择使用 Visual StudioVisual Studio for Mac 或 Visual Studio Code 等开发环境。有关详细信息，请参阅 .NET Core 入门。
- 可在 Windows 上的 Visual Studio 中使用 EF Core 来开发应用程序。建议使用最新版本的 Visual Studio。
- EF Core 可以在 Xamarin 和 .NET Native 等其他 .NET 实现上运行。但在实践中，这些实现具有运行时限制，可能会影响 EF Core 处理应用的效率。有关详细信息，请参阅 EF Core 支持的 .NET 实现。
- 最后，不同的数据库提供程序可能需要特定的数据库引擎版本、.NET 实现或操作系统。请确保可用的 EF Core 数据库提供程序支持适用于应用程序的环境。

获取 Entity Framework Core 运行时

要将 EF Core 添加到应用程序，请安装适用于要使用的数据库提供程序的 NuGet 包。

如果要生成 ASP.NET Core 应用程序，不需要安装内存中和 SQL Server 提供程序。这些提供程序随 EF Core 运行时一起包含在当前版本的 ASP.NET Core 中。

要安装或更新 NuGet 包，可以使用 .NET Core 命令行界面 (CLI)、Visual Studio 包管理器对话框或 Visual Studio 包管理器控制台。

.NET Core CLI

- 在操作系统的命令行中使用以下 .NET Core CLI 命令来安装或更新 EF Core SQL Server 提供程序：

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

- 可以使用 `dotnet add package` 修饰符在 `-v` 命令中指明特定的版本。例如，若要安装 EF Core 2.2.0 包，请将 `-v 2.2.0` 追加到命令中。

有关详细信息，请参阅 [.NET 命令行接口 \(CLI\) 工具](#)。

Visual Studio NuGet 包管理器对话框

- 从 Visual Studio 菜单中选择“项目”>“管理 NuGet 包”
- 单击“浏览”或“更新”选项卡
- 若要安装或更新 SQL Server 提供程序，请选择 `Microsoft.EntityFrameworkCore.SqlServer` 包并确认。

有关详细信息，请参阅 [NuGet 包管理器对话框](#)。

Visual Studio NuGet 包管理器控制台

- 从 Visual Studio 菜单中选择“工具”>“NuGet 包管理器”>“包管理器控制台”
- 若要安装 SQL Server 提供程序，请在包管理器控制台中运行以下命令：

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

- 若要更新提供程序，使用 `Update-Package` 命令。
- 若要指定特定版本，可以使用 `-Version` 修饰符。例如，若要安装 EF Core 2.2.0 包，请将 `-Version 2.2.0` 追加到命令中

有关详细信息，请参阅[包管理器控制台](#)。

获取 Entity Framework Core 工具

可以安装工具来执行项目中与 EF Core 相关的任务，例如创建和应用数据库迁移，或基于现有数据库创建 EF Core 模型。

提供了两个工具集：

- [.NET Core 命令行接口 \(CLI\) 工具](#)可用于 Windows、Linux 或 macOS。这些命令以 `dotnet ef` 开头。
- [包管理器控制台 \(PMC\) 工具](#)在 Windows 上的 Visual Studio 中运行。这些命令以动词开头，例如 `Add-Migration`、`Update-Database`。

虽然也可在包管理器控制台中使用 `dotnet ef` 命令，但在使用 Visual Studio 时建议使用包管理器控制台工具：

- 它们会自动使用在 Visual Studio 的 PMC 中选择的当前项目，无需手动切换目录。
- 命令完成后，它们会自动在 Visual Studio 中打开命令所生成的文件。

获取 .NET Core CLI 工具

.NET core CLI 工具需要前面的[系统必备](#)中提到的 .NET Core SDK。

`dotnet ef` 命令包含在当前版本的 .NET Core SDK 中，但若要对特定命令启用该命令，必须安装 `Microsoft.EntityFrameworkCore.Design` 包：

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

IMPORTANT

请务必使用与运行时包主版本匹配的工具包版本。

获取包管理器控制台工具

若要获取适用于 EF Core 的包管理器控制台工具，请安装 `Microsoft.EntityFrameworkCore.Tools` 包。例如，在 Visual Studio 中：

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

ASP.NET Core 应用自动随附此包。

升级到最新版本的 EF Core

- 每次发布新版本的 EF Core 时，我们都还会在 EF Core 项目中发布新版本的提供程序，如 `Microsoft.EntityFrameworkCore.SqlServer`、`Microsoft.EntityFrameworkCore.Sqlite` 和 `Microsoft.EntityFrameworkCore.InMemory`。只需升级到新版本的提供程序即可获取所有改进。
- EF Core 以及 SQL Server 和内存中提供程序包含在当前版本的 ASP.NET Core 中。若要将现有 ASP.NET Core

应用程序升级到较新版本的 EF Core, 请务必升级 ASP.NET Core 的版本。

- 如需更新使用第三方数据库提供程序的应用程序, 请始终检查与要使用的 EF Core 版本兼容的提供程序有无更新。例如, 早期版本的数据库提供程序不兼容 2.0 版 EF Core 运行时。
- 用于 EF Core 的第三方提供程序通常不随 EF Core 运行发布修补程序版本。若要将使用第三方提供程序的应用程序升级到 EF Core 的修补程序版本, 可能需要添加对单独的 EF Core 运行时组件(如 Microsoft.EntityFrameworkCore 和 Microsoft.EntityFrameworkCore.Relational)的直接引用。
- 若要将现有应用程序升级到最新版本的 EF Core, 可能需要手动删除一些对旧版 EF Core 包的引用:
 - EF Core 2.0 及更高版本不再需要或支持 `Microsoft.EntityFrameworkCore.SqlServer.Design` 等数据库提供程序设计时包, 但在升级其他包后, 它们不会被自动删除。
 - 自版本 2.1 起的 .NET SDK 包含 .NET CLI 工具, 这样就可以从项目文件中删除对相应包的引用:

```
<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
```

连接字符串

2020/3/11 ·

大多数数据库提供程序都需要某种形式的连接字符串才能连接到数据库。有时，此连接字符串包含需要保护的敏感信息。在开发、测试和生产环境等环境之间移动应用程序时，可能还需要更改连接字符串。

WinForms & WPF 应用程序

WinForms、WPF 和 ASP.NET 4 应用程序都有一个已尝试并经过测试的连接字符串模式。如果你使用的是 ASP.NET，则应将连接字符串添加到应用程序的 app.config 文件(web.config)中。如果您的连接字符串包含敏感信息(例如用户名和密码)，则可以使用[机密管理器工具](#)来保护配置文件的内容。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <connectionStrings>
    <add name="BloggingDatabase"
         connectionString="Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;" />
  </connectionStrings>
</configuration>
```

TIP

由于数据库提供程序是通过代码配置的，因此在 App.config 中存储的 EF Core 连接字符串上不需要 providerName 设置。

然后，可以使用上下文的 `OnConfiguring` 方法中的 `ConfigurationManager` API 读取连接字符串。可能需要添加对 `System.Configuration` framework 程序集的引用才能使用此 API。

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {

        optionsBuilder.UseSqlServer(ConfigurationManager.ConnectionStrings["BloggingDatabase"].ConnectionString);
    }
}
```

通用 Windows 平台 (UWP)

UWP 应用程序中的连接字符串通常是仅指定本地文件名的 SQLite 连接。它们通常不包含敏感信息，并且在部署应用程序时无需更改。因此，这些连接字符串通常可保留在代码中，如下所示。如果希望将它们移出代码，则 UWP 支持设置概念，有关详细信息，请参阅[uwp 文档的 "应用设置" 部分](#)。

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blogging.db");
    }
}
```

ASP.NET Core

在 ASP.NET Core 配置系统非常灵活，并且可以将连接字符串存储在 `appsettings.json`、环境变量、用户密钥存储或其他配置源中。有关更多详细信息，请参阅[ASP.NET Core 文档](#)的“配置”部分。下面的示例演示 `appsettings.json` 中存储的连接字符串。

```
{
  "ConnectionStrings": {
    "BloggingDatabase": "Server=(localdb)\\mssqllocaldb;Database=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;"
  },
}
```

上文通常在 `Startup.cs` 中进行配置，其中的连接字符串是从配置中读取的。请注意，`GetConnectionString()` 方法查找其键 `ConnectionStrings:<connection string name>` 的配置值。需要导入 Microsoft `extension.Configuration` 命名空间才能使用此扩展方法。

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("BloggingDatabase")));
}
```

日志记录

2020/3/11 •

TIP

可在 GitHub 上查看此文章的示例。

ASP.NET Core 应用程序

使用 `AddDbContext` 或 `AddDbContextPool` 时, EF Core 会自动与 ASP.NET Core 的日志记录机制集成。因此, 在使用 ASP.NET Core 时, 应按[ASP.NET Core 文档](#)中所述配置日志记录。

其他应用程序

EF Core 日志记录要求使用一个或多个日志记录提供程序配置的 `ILoggerFactory`。以下包中随附有常见提供程序:

- `""`。[控制台](#): 一个简单的控制台记录器。
- [AzureAppServices](#): 支持 Azure 应用服务 "诊断日志" 和 "日志流" 功能。
- 使用 `System.Exception()` 将日志记录到调试器监视器中的日志[记录](#)。
- 对 Windows 事件日志的日志[记录](#)。事件日志。
- `EventListener`: 支持 `EventSource/`。
- `TraceSource`: 使用 `System.Diagnostics.TraceSource.TraceEvent()` 将日志记录到跟踪倾听器。

安装适当的包后, 应用程序应创建 `Server.LoggerFactory` 的单一实例/全局实例。例如, 使用控制台记录器:

- [3.x 版](#)
- [版本 2.x](#)

```
public static readonly ILoggerFactory MyLoggerFactory
    = LoggerFactory.Create(builder => { builder.AddConsole(); });
```

然后, 应向 `DbContextOptionsBuilder` 上的 EF Core 注册此单一实例/全局实例。例如:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLoggerFactory(MyLoggerFactory) // Warning: Do not create a new ILoggerFactory instance each time
        .UseSqlServer(
            @"Server=(localdb)\mssqllocaldb;Database=EFLogging;Trusted_Connection=True;ConnectRetryCount=0");
```

WARNING

应用程序不应为每个上下文实例创建新的 `ILoggerFactory` 实例, 这一点非常重要。这样做会导致内存泄漏和性能下降。

筛选记录内容

应用程序可以通过在 `ILoggerProvider` 上配置筛选器来控制要记录的内容。例如:

- [3.x 版](#)

- 版本 2.x

```
public static readonly ILoggerFactory MyLoggerFactory
= LoggerFactory.Create(builder =>
{
    builder
        .AddFilter((category, level) =>
            category == DbLoggerCategory.Database.Command.Name
            && level == LogLevel.Information)
        .AddConsole();
});
```

在此示例中，将筛选日志以仅返回消息：

- 在 "Microsoft.entityframeworkcore" 类别中
- 在 "信息" 级别

对于 EF Core，记录器类别在 `DbLoggerCategory` 类中定义，以方便查找类别，但这些类别解析为简单字符串。

有关基础日志记录基础结构的更多详细信息，请参阅[ASP.NET Core 日志记录文档](#)。

连接复原

2020/3/11 •

连接复原将自动重试已失败的数据库命令。通过提供“执行策略”，它封装检测故障，然后重试命令所需的逻辑，该功能可以应用于任何数据库。EF Core 提供程序针对特定数据库的失败条件提供定制的执行策略，同时提供最佳的重试策略。

例如，SQL Server 提供程序包括专门针对 SQL Server（包括 SQL Azure）定制的执行策略。它可以识别可重试的异常类型，并具有可用于最大重试次数、重试间隔时间等的合理默认值。

为上下文配置选项时将指定执行策略。这通常位于派生上下文的 `OnConfiguring` 方法中：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=
(localdb)\mssqllocaldb;Database=EFMiscellaneous.ConnectionResiliency;Trusted_Connection=True;ConnectRetryCount=0",
            options => options.EnableRetryOnFailure());
}
```

或在 ASP.NET Core 应用程序的 `Startup.cs` 中：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<PicnicContext>(
        options => options.UseSqlServer(
            "<connection string>",
            providerOptions => providerOptions.EnableRetryOnFailure()));
}
```

自定义执行策略

如果要更改任何默认值，则可以使用一种机制来注册自己的自定义执行策略。

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseMyProvider(
            "<connection string>",
            options => options.ExecutionStrategy(...));
}
```

执行策略和事务

在出现故障时自动重试的执行策略需要能够回滚失败的重试块中的每个操作。启用重试后，通过 EF Core 执行的每个操作都将成为其自身的可重试操作。也就是说，如果发生暂时性故障，每个查询和对 `SaveChanges()` 的每个调用都将作为一个单元重试。

但是，如果您的代码使用来启动事务 `BeginTransaction()` 您要定义自己的需要被视为一个单元的一组操作，则需要播放该事务中的所有内容。如果尝试在使用执行策略时执行此操作，将收到如下所示的异常：

InvalidOperationException: 配置的执行策略 "SqlServerRetryingExecutionStrategy" 不支持用户启动的事务。使用由 "DbContext.Database.CreateExecutionStrategy()" 返回的执行策略执行事务(作为一个可回溯单元)中的所有操作。

解决方法是使用代表需要执行的所有内容的委托来手动调用执行策略。如果出现暂时性失败，执行策略将再次调用委托。

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var context = new BloggingContext())
        {
            using (var transaction = context.Database.BeginTransaction())
            {
                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/dotnet"});
                context.SaveChanges();

                context.Blogs.Add(new Blog {Url = "http://blogs.msdn.com/visualstudio"});
                context.SaveChanges();

                transaction.Commit();
            }
        });
    });
}
```

此方法还可用于环境事务。

```
using (var context1 = new BloggingContext())
{
    context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });

    var strategy = context1.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var context2 = new BloggingContext())
        {
            using (var transaction = new TransactionScope())
            {
                context2.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context2.SaveChanges();

                context1.SaveChanges();

                transaction.Complete();
            }
        });
    });
}
```

事务提交失败和幂等性问题

通常，如果连接失败，当前事务将回滚。但是，如果在提交事务时断开连接，则事务的生成状态是未知的。

默认情况下，执行策略将重试该操作，就像事务已回滚一样，但如果不是这样，则这会导致在新数据库状态不兼容时导致异常，或者如果操作不依赖于特定状态(例如，使用自动生成的键值插入新行时)，可能会导致数据损坏。

可以通过多种方式来处理此情况。

选项 1-执行(几乎不)任何操作

事务提交期间连接失败的可能性很低, 因此如果实际发生此情况, 应用程序可能会失败。

但是, 需要避免使用存储生成的密钥, 以确保引发异常而不是添加重复的行。请考虑使用客户端生成的 GUID 值或客户端值生成器。

选项 2-重建应用程序状态

1. 放弃当前 `DbContext`。
2. 创建新 `DbContext`, 并从数据库中还原应用程序的状态。
3. 通知用户上次操作可能尚未成功完成。

选项 3-添加状态验证

对于大多数更改数据库状态的操作, 可以添加代码来检查它是否成功。EF 提供扩展方法, 使其更易于

`IExecutionStrategy.ExecuteInTransaction`。

此方法将启动并提交事务, 并且还接受 `verifySucceeded` 参数中的一个函数, 该参数会在事务提交期间发生暂时性错误时调用。

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    var blogToAdd = new Blog {Url = "http://blogs.msdn.com/dotnet"};
    db.Blogs.Add(blogToAdd);

    strategy.ExecuteInTransaction(db,
        operation: context =>
    {
        context.SaveChanges(acceptAllChangesOnSuccess: false);
    },
    verifySucceeded: context => context.Blogs.AsNoTracking().Any(b => b.BlogId == blogToAdd.BlogId));

    db.ChangeTracker.AcceptAllChanges();
}
```

NOTE

此处 `SaveChanges` 在 `acceptAllChangesOnSuccess` 设置为 `false` 时调用, 以避免 `Unchanged` 成功时将 `Blog` 实体的状态更改为 `SaveChanges`。如果提交失败并且事务已回滚, 则允许重试相同的操作。

选项 4-手动跟踪事务

如果需要使用存储生成的密钥或需要一种处理提交失败且不依赖于所执行操作的通用方法, 则可以为每个事务分配一个可在提交失败时进行检查的 ID。

1. 向数据库添加一个用于跟踪事务状态的表。
2. 在每个事务开头的表中插入一行。
3. 如果在提交期间连接失败, 请检查数据库中是否存在相应的行。
4. 如果提交成功, 则删除相应的行以避免表增长。

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });

    var transaction = new TransactionRow { Id = Guid.NewGuid() };
    db.Transactions.Add(transaction);

    strategy.ExecuteInTransaction(db,
        operation: context =>
    {
        context.SaveChanges(acceptAllChangesOnSuccess: false);
    },
        verifySucceeded: context => context.Transactions.AsNoTracking().Any(t => t.Id == transaction.Id));

    db.ChangeTracker.AcceptAllChanges();
    db.Transactions.Remove(transaction);
    db.SaveChanges();
}
```

NOTE

请确保用于验证的上下文具有定义的执行策略，因为如果在事务提交期间连接失败，则连接可能在验证期间再次失败。

测试使用 EF Core 的代码

2020/4/9 ·

测试访问数据库的代码需要满足以下任一条件：

- 针对在生产环境中使用的相同数据库系统运行查询和更新。
- 对其他更易管理的数据库系统运行查询和更新。
- 使用测试替身或其他机制来避免使用数据库。

本文档概述了每个选项所涉及的折衷方案，并说明了如何在每种方法中使用 EF Core。

所有数据库提供程序都不等同

务必要理解的是，EF Core 不是为了抽象底层数据库系统的各个方面而设计的。EF Core 是一组通用的模式和概念，可用于任何数据库系统。然后，EF Core 数据库提供程序在此通用框架的基础上叠加数据库特定的行为和功能。这样，每个数据库系统都可以实现其最佳性能，并在适当的情况下维持与其他数据库系统的共性。

从根本上说，这意味着切换数据库提供程序将更改 EF Core 行为，使应用程序不能正常工作，除非它明确解释行为的所有差异。虽然如此，但在许多情况下还是可以这么做，因为关系数据库之间存在高度的共性。这有利也有弊。其有利之处在于，可以相对轻松地切换数据库。而其弊端在于，如果未针对新的数据库系统彻底测试应用程序，会有一种虚假的安全感。

方法 1：生产数据库系统

如前一部分所述，要确保测试在生产中运行的内容，唯一的方法是使用同一个数据库系统。例如，如果部署的应用程序使用 SQL Azure，则测试也应针对 SQL Azure 进行。

但是，要让每个开发者正在积极处理代码的同时针对 SQL Azure 运行测试，不仅工作速度慢而且代价高昂。以下问题描述了这些方法中涉及的主要折衷方案：什么时候适合偏离生产数据库系统以提高测试效率？

幸运的是，在本例中，答案非常简单：使用本地 SQL Server 进行开发者测试。SQL Azure 和 SQL Server 极其相似，因此针对 SQL Server 的测试通常是合理的折衷方案。虽然如此，但在投入生产之前针对 SQL Azure 本身运行测试仍然是明智之举。

LocalDb

所有主要数据库系统都具有某种形式的“Developer Edition”本地测试。SQL Server 还有一项名为 [LocalDb](#) 的功能。LocalDb 的主要优势在于可以按需增大数据库实例。这可以避免在计算机上运行数据库服务，即使在未运行测试时也是如此。

LocalDb 也不是没有问题：

- 它不支持 [SQL Server Developer Edition](#) 支持的部分内容。
- 它在 Linux 上不可用。
- 由于要启动服务，可能导致首次测试运行滞后。

就个人而言，我从来不觉得在开发计算机上运行数据库服务有什么问题，所以一般建议使用 Developer Edition。但是，这种做法对某些人而言可能适用，尤其是在处理能力不够强的开发计算机上。

方法 2：SQLite

EF Core 主要通过针对本地 SQL Server 实例运行 SQL Server 提供程序来对其进行测试。这些测试将在几分钟内在一台高速计算机上运行数万次查询。这说明，使用实际数据库系统可能是一种高性能的解决方案。有一种错误的

观念是，使用较小型的数据库是快速运行测试的唯一方法。

话虽如此，但如果由于某些原因，无法针对接近生产数据库系统的系统运行测试，应该怎么办？下一个最佳选择是使用具有相似功能的对象。这通常意味着另一个关系数据库，[SQLite](#) 是显而易见的选择。

SQLite 是不错的选择，因为：

- 它在处理应用程序的过程中运行，因此开销较低。
- 它使用自动创建的简单数据库文件，因此不需要数据库管理。
- 它有内存模式，甚至可以避免创建文件。

但是，请记住：

- SQLite 不一定支持你的生产数据库系统支持的所有内容，这是不可避免的。
- 对于某些查询，SQLite 的行为方式将与你的生产数据库系统不同。

因此，如果使用 SQLite 进行测试，还应确保针对实际数据库系统进行测试。

有关 EF Core 特定指南，请参阅[用 SQLite 进行测试](#)。

方法 3: EF Core 内存中数据库

EF Core 附带了内存中数据库，用于对 EF Core 本身进行内部测试。此数据库一般不适合用于测试使用 EF Core 的应用程序。尤其是在下列情况下：

- 它不是关系数据库
- 它不支持事务
- 它未针对性能进行优化

在测试 EF Core 内部机制时，这些都不重要，因为我们只在数据库与测试不相关时才会使用它。另一方面，在测试使用 EF Core 的应用程序时，这些特性往往非常重要。

单元测试

考虑这种情况：你要测试可能需要使用数据库中的某些数据的业务逻辑，但该测试本身并不测试数据库交互。可以选择使用[测试替身](#)，例如模拟或虚构数据库。

我们使用测试替身进行 EF Core 的内部测试。但是，我们不会尝试模拟 DbContext 或 IQueryable。这样做难度大、过程繁琐且结果不可靠。不要这样做。

改为使用内存中数据库对使用 DbContext 的应用进行单元测试。在这种情况下，使用内存中数据库是适当的，因为测试不依赖于数据库行为。但是不要这样测试实际数据库查询或更新。

请参阅[使用内存中提供程序进行测试](#)，获取 EF Core 特定指导，了解如何使用内存中数据库进行单元测试。

使用 SQLite 进行测试

2020/3/11 ·

SQLite 具有内存中模式，借此可使用 SQLite 针对关系数据库编写测试，而不会产生实际数据库操作的开销。

TIP

你可以在 GitHub 上查看此文章的[示例](#)

示例测试方案

请考虑以下允许应用程序代码执行一些与博客相关的操作的服务。在内部，它使用连接到 SQL Server 数据库的 `DbContext`。交换此上下文可有效连接到内存中 SQLite 数据库，这样我们就可以为该服务编写高效的测试，而无需修改代码或执行大量的工作来创建上下文的测试副本。

```
using System.Collections.Generic;
using System.Linq;

namespace BusinessLogic
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public void Add(string url)
        {
            var blog = new Blog { Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();
        }

        public IEnumerable<Blog> Find(string term)
        {
            return _context.Blogs
                .Where(b => b.Url.Contains(term))
                .OrderBy(b => b.Url)
                .ToList();
        }
    }
}
```

准备您的上下文

避免配置两个数据库提供程序

在测试中，从外部配置上下文以使用 `InMemory` 提供程序。如果通过在上下文中重写 `OnConfiguring` 来配置数据库提供程序，则需要添加一些条件代码，以确保仅配置数据库提供程序（如果尚未配置）。

TIP

如果使用的是 ASP.NET Core，则不需要此代码，因为数据库提供程序是在上下文之外（在 Startup.cs 中）配置的。

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}
```

添加用于测试的构造函数

对不同数据库启用测试的最简单方法是修改上下文以公开接受 `DbContextOptions<TContext>` 的构造函数。

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    { }

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }
```

TIP

`DbContextOptions<TContext>` 告知上下文所有设置，如要连接到的数据库。这与在上下文中运行 `OnConfiguring` 方法所生成的对象相同。

编写测试

使用此提供程序进行测试的关键是，可告知上下文使用 SQLite 并控制内存中数据库的范围。通过打开和关闭连接来控制数据库的范围。数据库的范围限定为连接打开的持续时间。通常，每个测试方法都需要一个干净的数据库。

TIP

若要使用 `SqliteConnection()` 和 `.UseSqlite()` 扩展方法，请参考 NuGet 包 [microsoft.entityframeworkcore](#)。

```
using BusinessLogic;
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using Xunit;

namespace EFTesting.TestProject.SQLite
{
    public class BlogServiceTests
    {
        [Fact]
        public void Add_writes_to_database()
        {
            // In-memory database only exists while the connection is open
            var connection = new SqliteConnection("DataSource=:memory:");
            connection.Open();
```

```

try
{
    var options = new DbContextOptionsBuilder<BloggingContext>()
        .UseSqlite(connection)
        .Options;

    // Create the schema in the database
    using (var context = new BloggingContext(options))
    {
        context.Database.EnsureCreated();
    }

    // Run the test against one instance of the context
    using (var context = new BloggingContext(options))
    {
        var service = new BlogService(context);
        service.Add("https://example.com");
        context.SaveChanges();
    }

    // Use a separate instance of the context to verify correct data was saved to database
    using (var context = new BloggingContext(options))
    {
        Assert.Equal(1, context.Blogs.Count());
        Assert.Equal("https://example.com", context.Blogs.Single().Url);
    }
}
finally
{
    connection.Close();
}

[Fact]
public void Find_searches_url()
{
    // In-memory database only exists while the connection is open
    var connection = new SqliteConnection("DataSource=:memory:");
    connection.Open();

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlite(connection)
            .Options;

        // Create the schema in the database
        using (var context = new BloggingContext(options))
        {
            context.Database.EnsureCreated();
        }

        // Insert seed data into the database using one instance of the context
        using (var context = new BloggingContext(options))
        {
            context.Blogs.Add(new Blog { Url = "https://example.com/cats" });
            context.Blogs.Add(new Blog { Url = "https://example.com/catfish" });
            context.Blogs.Add(new Blog { Url = "https://example.com/dogs" });
            context.SaveChanges();
        }

        // Use a clean instance of the context to run the test
        using (var context = new BloggingContext(options))
        {
            var service = new BlogService(context);
            var result = service.Find("cat");
            Assert.Equal(2, result.Count());
        }
    }
}

```

```
        finally
        {
            connection.Close();
        }
    }
}
```

使用 InMemory 进行测试

2020/3/11 ·

如果要使用与连接到实际数据库类似的内容来测试组件，而不是实际数据库操作的开销，则 InMemory 提供程序很有用。

TIP

可在 GitHub 上查看此文章的[示例](#)。

InMemory 不是关系数据库

EF Core 数据库提供程序不需要是关系数据库。InMemory 旨在作为一般用途的数据库进行测试，而不是为模拟关系数据库而设计。

其中的一些示例包括：

- InMemory 允许您保存违反关系数据库中的引用完整性约束的数据。
- 如果对模型中的属性使用 `DefaultValueSql (string)`，则这是一个关系数据库 API，在对 InMemory 运行时将不起作用。
- 不支持[通过时间戳/行版本](#) (`[Timestamp]` 或 `IsRowVersion`) 的并发。如果使用旧并发标记完成更新，将不会引发[DbUpdateConcurrencyException](#)。

TIP

很多测试目的都不重要。但是，如果您想要对行为更像是真实关系数据库的某些内容进行测试，则应考虑使用[SQLite 内存中模式](#)。

示例测试方案

请考虑以下允许应用程序代码执行一些与博客相关的操作的服务。在内部，它使用连接到 SQL Server 数据库的 `DbContext`。交换此上下文以连接到 InMemory 数据库会很有用，这样就可以为此服务编写高效的测试，而无需修改代码，或执行大量工作来创建上下文的测试双精度。

```
using System.Collections.Generic;
using System.Linq;

namespace BusinessLogic
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public void Add(string url)
        {
            var blog = new Blog { Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();
        }

        public IEnumerable<Blog> Find(string term)
        {
            return _context.Blogs
                .Where(b => b.Url.Contains(term))
                .OrderBy(b => b.Url)
                .ToList();
        }
    }
}
```

准备您的上下文

避免配置两个数据库提供程序

在测试中，从外部配置上下文以使用 InMemory 提供程序。如果通过在上下文中重写 `OnConfiguring` 来配置数据库提供程序，则需要添加一些条件代码，以确保仅配置数据库提供程序（如果尚未配置）。

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFProviders.InMemory;Trusted_Connection=True;ConnectRetryCount=0");
    }
}
```

TIP

如果你使用的 ASP.NET Core，则应不需要此代码由于外部（会在 `Startup.cs` 的上下文已配置数据库提供程序）。

添加用于测试的构造函数

对不同数据库启用测试的最简单方法是修改上下文以公开接受 `DbContextOptions<TContext>` 的构造函数。

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    { }

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }
```

TIP

`DbContextOptions<TContext>` 告知上下文所有设置，如要连接到的数据库。这与在上下文中运行 `OnConfiguring` 方法所生成的对象相同。

编写测试

使用此提供程序进行测试的关键是能够通知上下文使用 `InMemory` 提供程序，并控制内存中数据库的范围。通常，每个测试方法都需要一个干净的数据库。

下面是使用 `InMemory` 数据库的测试类的示例。每个测试方法均指定唯一的数据库名称，也就是说，每个方法都有其自己的 `InMemory` 数据库。

TIP

若要使用 `.UseInMemoryDatabase()` 扩展方法，请参考 NuGet 包[microsoft.entityframeworkcore](#)。

```

using BusinessLogic;
using Microsoft.EntityFrameworkCore;
using System.Linq;
using Xunit;

namespace EFTesting.TestProject.InMemory
{
    public class BlogServiceTests
    {
        [Fact]
        public void Add_writes_to_database()
        {
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseInMemoryDatabase(databaseName: "Add_writes_to_database")
                .Options;

            // Run the test against one instance of the context
            using (var context = new BloggingContext(options))
            {
                var service = new BlogService(context);
                service.Add("https://example.com");
                context.SaveChanges();
            }

            // Use a separate instance of the context to verify correct data was saved to database
            using (var context = new BloggingContext(options))
            {
                Assert.Equal(1, context.Blogs.Count());
                Assert.Equal("https://example.com", context.Blogs.Single().Url);
            }
        }

        [Fact]
        public void Find_searches_url()
        {
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseInMemoryDatabase(databaseName: "Find_searches_url")
                .Options;

            // Insert seed data into the database using one instance of the context
            using (var context = new BloggingContext(options))
            {
                context.Blogs.Add(new Blog { Url = "https://example.com/cats" });
                context.Blogs.Add(new Blog { Url = "https://example.com/catfish" });
                context.Blogs.Add(new Blog { Url = "https://example.com/dogs" });
                context.SaveChanges();
            }

            // Use a clean instance of the context to run the test
            using (var context = new BloggingContext(options))
            {
                var service = new BlogService(context);
                var result = service.Find("cat");
                Assert.Equal(2, result.Count());
            }
        }
    }
}

```

配置 DbContext

2020/3/25 ·

本文介绍了使用特定 EF Core 提供程序和可选行为通过 `DbContextOptions` 配置 `DbContext` 的基本模式。

设计时 DbContext 配置

EF Core 的设计时工具需要能够发现和创建 `DbContext` 类型的工作实例，以便收集有关该应用程序的实体类型及其如何映射到数据库架构的详细信息。只要该工具可以轻松地创建 `DbContext`，就可以自动执行此过程，因为它的配置方式与在运行时的配置方式类似。

虽然向 `DbContext` 提供必要的配置信息的任何模式在运行时都可以使用，但在设计时需要使用 `DbContext` 的工具只能处理有限数量的模式。[设计时上下文创建部分](#)更详细地介绍了这些内容。

配置 DbContextOptions

`DbContext` 必须具有 `DbContextOptions` 的实例才能执行任何工作。`DbContextOptions` 实例携带如下配置信息：

- 要使用的数据库提供程序，通常通过调用方法（如 `UseSqlServer` 或 `UseSqlite`）进行选择。这些扩展方法需要相应的提供程序包，如 `Microsoft.EntityFrameworkCore.SqlServer` 或 `Microsoft.EntityFrameworkCore.Sqlite`。方法在 `Microsoft.EntityFrameworkCore` 命名空间中定义。
- 任何必需的数据库实例的连接字符串或标识符，通常作为参数传递给上面提到的提供者选择方法
- 任何提供程序级别的可选行为选择器，通常还链接到对提供程序选择方法的调用中
- 任何常规 EF Core 行为选择器，通常链接之后或之前提供程序选择器方法

下面的示例将 `DbContextOptions` 配置为使用 SQL Server 提供程序、包含在 `connectionString` 变量中的连接、提供程序级别的命令超时，以及在默认情况下 `DbContext` 执行所有查询的 EF Core 行为选择器：

```
optionsBuilder
    .UseSqlServer(connectionString, providerOptions=>providerOptions.CommandTimeout(60))
    .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
```

NOTE

提供程序选择器方法和上面提到的其他行为选择器方法是 `DbContextOptions` 或特定于提供程序的选项类的扩展方法。若要访问这些扩展方法，你可能需要在范围内具有命名空间（通常为 `Microsoft.EntityFrameworkCore`），并在项目中包含附加包依赖项。

可以通过重写 `OnConfiguring` 方法或通过构造函数参数从外部来向 `DbContext` 提供 `DbContextOptions`。

如果同时使用这两个，则 `OnConfiguring` 最后应用，并且可以覆盖提供给构造函数参数的选项。

构造函数参数

构造函数可以简单地接受 `DbContextOptions`，如下所示：

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

TIP

DbContext 的基本构造函数还接受 `DbContextOptions` 的非泛型版本，但对于具有多个上下文类型的应用程序，不建议使用非泛型版本。

应用程序现在可以在实例化上下文时传递 `DbContextOptions`，如下所示：

```
var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))
{
    // do stuff
}
```

OnConfiguring

您还可以在上下文本身中初始化 `DbContextOptions`。虽然你可以将此方法用于基本配置，但你通常仍需要从外部获取某些配置详细信息，例如数据库连接字符串。可以使用配置 API 或其他任何方法来完成此操作。

若要在上下文中初始化 `DbContextOptions`，请重写 `OnConfiguring` 方法，并对提供的 `DbContextOptionsBuilder` 调用方法：

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }
}
```

应用程序可以简单地实例化此类上下文，而无需将任何内容传递给构造函数：

```
using (var context = new BloggingContext())
{
    // do stuff
}
```

TIP

此方法不会对其进行测试，除非测试以完整数据库为目标。

将 `DbContext` 与依赖关系注入一起使用

EF Core 支持将 `DbContext` 与依赖关系注入容器一起使用。可以通过使用 `AddDbContext<DbContext>` 方法将

DbContext 类型添加到服务容器中。

AddDbContext<TContext> 将使你的 DbContext 类型、 TContext 和对应的 DbContextOptions<TContext> 可用于从服务容器中注入。

有关依赖关系注入的其他信息, [请参阅](#)下文。

将 DbContext 添加到依赖关系注入:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options => options.UseSqlite("Data Source=blog.db"));
}
```

这需要将[构造函数参数](#)添加到接受 DbContextOptions<TContext> 的 DbContext 类型。

上下文代码:

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        :base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

(在 ASP.NET Core) 的应用程序代码:

```
public class MyController
{
    private readonly BloggingContext _context;

    public MyController(BloggingContext context)
    {
        _context = context;
    }

    ...
}
```

应用程序代码(直接使用 ServiceProvider, 不太常见):

```
using (var context = serviceProvider.GetService<BloggingContext>())
{
    // do stuff
}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>();
```

避免 DbContext 线程问题

Entity Framework Core 不支持在同一个 DbContext 实例上运行多个并行操作。这包括异步查询的并行执行以及从多个线程进行的任何显式并发使用。因此, 始终 await 异步调用, 或对并行执行的操作使用单独的 DbContext 实例。

当 EF Core 检测到同时尝试使用 DbContext 实例时, 你将看到一条 InvalidOperationException, 其中包含类似于下面的消息:

在上一个操作完成之前，在此上下文上启动的第二个操作。这通常是由使用同一个 `DbContext` 实例的不同线程引起的，但不保证实例成员是线程安全的。

并发访问未被检测时，可能会导致未定义的行为、应用程序崩溃和数据损坏。

存在一些常见错误，可能会无意中导致同一 `DbContext` 实例的并发访问：

在对同一 `DbContext` 启动任何其他操作之前，忘记等待异步操作完成

使用异步方法，EF Core 可以启动以非阻止方式访问数据库的操作。但是，如果调用方不等待其中某个方法完成，并继续对 `DbContext` 执行其他操作，则 `DbContext` 的状态可能会损坏（并且很可能会损坏）。

始终等待立即 EF Core 异步方法。

通过依赖关系注入在多个线程之间隐式共享 `DbContext` 实例

默认情况下，`AddDbContext` 扩展方法会将 `DbContext` 类型注册为**范围生存期**。

这对于大多数 ASP.NET Core 应用程序中的并发访问问题是安全的，因为在给定的时间只有一个线程在执行每个客户端请求，因为每个请求都将获取一个单独的依赖项注入范围（因而单独的 `DbContext` 实例）。对于 Blazor 服务器托管模型，使用一个逻辑请求来维护 Blazor 用户线路，因此，如果使用默认注入作用域，则每个用户线路只能提供一个作用域内 `DbContext` 实例。

任何并行执行多个线程的代码都应确保 `DbContext` 实例不会同时访问。

使用依赖关系注入，这可以通过以下方式实现：将上下文注册为作用域，并为每个线程创建作用域（使用 `IServiceScopeFactory`），或将 `DbContext` 注册为暂时性（使用采用 `ServiceLifetime` 参数的 `AddDbContext` 的重载）。

阅读更多

- 有关使用 DI 的详细信息，请参阅[依赖关系注入](#)。
- 有关详细信息，请参阅[测试](#)。

使用可以为 null 的引用类型

2020/3/11 ·

C#8引入了一种名为 [null 的引用类型](#)的新功能，允许对引用类型进行批注，以指示它是否可用于包含 null。如果你不熟悉此功能，则建议你通过阅读C#文档来使自己熟悉该功能。

此页介绍 EF Core 对可为 null 的引用类型的 support，并介绍了使用它们的最佳做法。

必需属性和可选属性

对于必需属性和可选属性及其与可为 null 的引用类型的交互，主要文档是[必需的和可选的属性](#)页。建议首先阅读该页面。

NOTE

在现有项目上启用可以为 null 的引用类型时要格外小心：现在配置为可选的引用类型属性现在将配置为“必需”，除非它们显式批注为可为 null。管理关系数据库架构时，这可能会导致生成更改数据库列的为 null 性的迁移。

DbContext 和 DbSet

如果启用了可以为 null 的引用C#类型，则编译器将为任何未初始化的不可为 null 的属性发出警告，因为这将包含 null。因此，在上下文中定义不可为 null 的 `DbSet` 的常见做法现在会生成一个警告。但是，EF Core 始终初始化 `DbContext` 派生类型上的所有 `DbSet` 属性，因此即使编译器不知道此操作，也可以保证它们永远不会为 null。因此，建议保留不可以为 null 的 `DbSet` 属性，使你能够在不进行 null 检查的情况下访问这些属性，并通过使用 null 包容性运算符(!)的帮助将其显式设置为 null 来使编译器警告静音：

```
public class NullableReferenceTypesContext : DbContext
{
    public DbSet<Customer> Customers { get; set; } = null!;
    public DbSet<Order> Orders { get; set; } = null!;

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder
            .UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFNullableReferenceTypes;Trusted_Connection=True;ConnectRetryCount=0");
}
```

不可以为 null 的属性和初始化

对于实体类型上的常规属性，未初始化的不可以为 null 的引用类型的编译器警告也是一个问题。在上面的示例中，我们使用[构造函数绑定](#)避免了这些警告，这是一项完美使用不可为 null 的属性的功能，确保它们始终初始化。但是，在某些情况下，构造函数绑定不是一个选项：例如，不能以这种方式初始化导航属性。

所需的导航属性会带来额外的难度：尽管某个给定主体的依赖项始终存在，但特定查询可能会或不加载该依赖项，具体取决于程序中该点的需求（[请参阅不同的加载数据模式](#)）。同时，不需要将这些属性设置为可以为 null，因为这会强制对它们的所有访问权限，以检查是否有 null，即使它们是必需的。

处理这些方案的一种方法是使用一个可以为 null 的[支持字段](#)的不可为 null 的属性：

```
private Address? _shippingAddress;

public Address ShippingAddress
{
    set => _shippingAddress = value;
    get => _shippingAddress
        ?? throw new InvalidOperationException("Uninitialized property: " + nameof(ShippingAddress));
}
```

由于导航属性不可为 null，因此配置了必需的导航；只要正确加载导航，就可以通过属性访问依赖项。但是，如果在未事先正确加载相关实体的情况下访问属性，则会引发 InvalidOperationException，因为 API 协定的使用不正确。请注意，必须将 EF 配置为始终访问支持字段而不是属性，因为它依赖于即使在未设置时也能读取值；请参阅有关如何执行此操作的[支持字段](#)的文档，并考虑指定 `PropertyAccessMode.Field` 以确保配置正确。

作为 terser 的替代方法，可以使用包容性运算符(!)的帮助简单地将属性初始化为 null：

```
public Product Product { get; set; } = null!;
```

实际的 null 值永远不会被视为除了编程错误之外的情况，例如，访问导航属性时，无需事先正确加载相关实体。

NOTE

包含对多个相关实体的引用的集合导航应始终不可为 null。空集合意味着不存在相关实体，但列表本身不应为 null。

导航和包括可以为 null 的关系

当处理可选关系时，可能会遇到编译器警告，但不可能出现实际的 null 引用异常。在转换和执行 LINQ 查询时，EF Core 确保在一个可选的相关实体不存在的情况下，将忽略对该实体的任何导航，而不是引发。但是，编译器不知道这 EF Core 确保，并生成警告，就好像 LINQ 查询是在内存中执行的，而 LINQ to Objects。因此，需要使用包容性运算符(!)来通知编译器无法实现实际的 null 值：

```
Console.WriteLine(order.OptionalInfo!.ExtraAdditionalInfo!.SomeExtraAdditionalInfo);
```

在可选导航中包含多个级别的关系时，会发生类似的问题：

```
var order = context.Orders
    .Include(o => o.OptionalInfo!)
    .ThenInclude(op => op.ExtraAdditionalInfo)
    .Single();
```

如果你发现自己执行了很多操作，并且所涉及的实体类型是在 EF Core 查询中主要(或独占)使用的，请考虑使导航属性不可为 null，并将其配置为可通过熟知 API 或数据批注进行选择。这将删除所有编译器警告，同时保持关系为可选；但是，如果你的实体是在 EF Core 之外遍历的，则你可能会观察到 null 值，尽管这些属性已批注为不可为 null。

限制

- 反向工程目前不支持 [C# 8 个可以为 null 的引用类型\(NRTs\)](#)：C# EF Core 始终会生成假定该功能已关闭的代码。例如，可为 null 的文本列将被基架为具有类型 `string` 的属性，而不是 `string?`，其中使用的是用于配置是否需要属性的流畅 API 或数据批注。您可以编辑基架代码并将其替换为 C# 可为 null 的批注。[#15520](#) 的问题跟踪了可为 null 的引用类型的基架支持。
- EF Core 的公共 API 图面尚未批注为空性(公共 API 为 "在意")，这使得在 NRT 功能打开时使用它有时会很难

使用。这特别包括 EF Core 公开的异步 LINQ 运算符，如[FirstOrDefaultAsync](#)。我们计划为5.0 版本解决这一情况。

创建并配置模型

2020/4/8 •

Entity Framework 使用一组约定基于实体类的定义来构建模型。可指定其他配置以补充和/或替代约定的内容。

本文介绍可应用于面向任何数据存储的模型配置，以及面向任意关系数据库时可应用的配置。提供程序还可支持特定于具体数据存储的配置。有关提供程序特定配置的文档，请参阅 [数据库提供程序](#) 部分。

TIP

可在 GitHub 上查看此文章的 [示例](#)。

使用 fluent API 配置模型

可在派生上下文中覆写 `OnModelCreating` 方法，并使用 `ModelBuilder API` 来配置模型。此配置方法最为有效，并可在不修改实体类的情况下指定配置。Fluent API 配置具有最高优先级，并将替代约定和数据注释。

```
using Microsoft.EntityFrameworkCore;

namespace EFModeling.FluentAPI.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }

        #region Required
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>()
                .Property(b => b.Url)
                .IsRequired();
        }
        #endregion
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Url { get; set; }
    }
}
```

使用数据注释来配置模型

也可将特性(称为数据注释)应用于类和属性。数据注释会替代约定，但会被 Fluent API 配置替代。

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

namespace EFModeling.DataAnnotations.Required
{
    class MyContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    #region Required
    public class Blog
    {
        public int BlogId { get; set; }
        [Required]
        public string Url { get; set; }
    }
    #endregion
}
```

实体类型

2020/3/11 •

在上下文中包含一种类型的 DbSet，这意味着它包含在 EF Core 的模型中；我们通常将此类类型称为实体。EF Core 可以从/向数据库中读取和写入实体实例，如果使用的是关系数据库，EF Core 可以通过迁移为实体创建表。

在模型中包含类型

按照约定，在上下文中的 DbSet 属性中公开的类型作为实体包含在模型中。还包括在 `OnModelCreating` 方法中指定的实体类型，就像通过递归方式浏览其他发现的实体类型的导航属性找到的任何类型一样。

在下面的代码示例中，包含了所有类型：

- `Blog` 包含在内，因为它在上下文的 DbSet 属性中公开。
- 包含 `Post` 是因为它是通过 `Blog.Posts` 导航属性发现的。
- `AuditEntry`，因为它是在 `OnModelCreating` 中指定的。

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

从模型中排除类型

如果您不希望在模型中包含某一类型，则可以排除它：

- **数据注释**

- 熟知 API

```
[NotMapped]  
public class BlogMetadata  
{  
    public DateTime LoadedFromDatabase { get; set; }  
}
```

表名

按照约定，每个实体类型将设置为映射到与公开实体的 DbSet 属性同名的数据库表。如果给定实体不存在 DbSet，则使用类名称。

可以手动配置表名：

- 数据注释
- 熟知 API

```
[Table("blogs")]  
public class Blog  
{  
    public int BlogId { get; set; }  
    public string Url { get; set; }  
}
```

表架构

使用关系数据库时，表按约定在数据库的默认架构中创建。例如，Microsoft SQL Server 将使用 `dbo` 架构（SQLite 不支持架构）。

你可以配置要在特定架构中创建的表，如下所示：

- 数据注释
- 熟知 API

```
[Table("blogs", Schema = "blogging")]  
public class Blog  
{  
    public int BlogId { get; set; }  
    public string Url { get; set; }  
}
```

您还可以在模型级别定义 Fluent API 的默认架构，而不是为每个表指定架构：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)  
{  
    modelBuilder.HasDefaultSchema("blogging");  
}
```

请注意，设置默认架构也会影响其他数据库对象，如序列。

实体属性

2020/3/11 •

模型中的每个实体类型都具有一组属性，这些属性 EF Core 将从数据库中读取和写入数据。如果使用关系数据库，实体属性将映射到表列。

包含和排除的属性

按照约定，具有 getter 和 setter 的所有公共属性都将包括在模型中。

可以按如下所述排除特定属性：

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

列名

按照约定，使用关系数据库时，实体属性映射到与属性同名的表列。

如果希望使用不同的名称配置列，可以执行以下操作：

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    [Column("blog_id")]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

列数据类型

使用关系数据库时，数据库提供程序根据属性的 .NET 类型选择数据类型。它还会考虑其他元数据，如配置的最大长度、属性是否是主键的一部分等。

例如，SQL Server 将 `DateTime` 属性映射到 `datetime2(7)` 列，并将 `string` 属性映射到 `nvarchar(max)` 列（或用于 `nvarchar(450)` 用作键的属性）。

您还可以配置列，以便为列指定精确的数据类型。例如，下面的代码将 `Url` 配置为具有最大长度的非 unicode 字符串 `200` 并将 `Rating` 为 decimal，精度为 `5`，小数位数为 `2`：

- [数据注释](#)

- 熟知 API

```
public class Blog
{
    public int BlogId { get; set; }
    [Column(TypeName = "varchar(200)")]
    public string Url { get; set; }
    [Column(TypeName = "decimal(5, 2)")]
    public decimal Rating { get; set; }
}
```

最大长度

配置最大长度会向数据库提供程序提供有关要为给定属性选择的相应列数据类型的提示。最大长度仅适用于数组数据类型，如 `string` 和 `byte[]`。

NOTE

将数据传递到提供程序之前，实体框架不会执行任何最大长度验证。由提供程序或数据存储在适当时机进行验证。例如，如果以 SQL Server 为目标，超过最大长度将引发异常，因为基础列的数据类型不允许存储过多数据。

在下面的示例中，配置最大长度为500将导致在 SQL Server 上创建 `nvarchar(500)` 类型的列：

- 数据注释
- 熟知 API

```
public class Blog
{
    public int BlogId { get; set; }
    [MaxLength(500)]
    public string Url { get; set; }
}
```

必需属性和可选属性

如果属性对于包含 `null` 是有效的，则认为该属性是可选的。如果 `null` 不是要分配给属性的有效值，则将其视为必需属性。映射到关系数据库架构时，必需的属性将创建为不可为 `null` 的列，而可选属性则创建为可以为 `null` 的列。

约定

按照约定，.NET 类型可以包含 `null` 的属性将配置为可选，而 .NET 类型不能包含 `null` 的属性将根据需要进行配置。例如，具有 .NET 值类型（`int`、`decimal`、`bool` 等）的所有属性都是必需的，并且具有可为 `null` 的 .NET 值类型（`int?`、`decimal?`、`bool?` 等）的所有属性都配置为可选。

C#8引入了一个名为 [null 的引用类型](#)的新功能，该功能允许对引用类型进行批注，以指示它是否可用于包含空值。此功能在默认情况下处于禁用状态，如果启用，它将按以下方式修改 EF Core 的行为：

- 如果禁用了可为 `null` 的引用类型（默认值），则所有具有 .NET 引用类型的属性都将按约定（例如 `string`）配置为可选。
- 如果启用了可以为 `null` 的引用类型，则将根据 .NET C#类型的为 `null` 性配置属性：`string?` 将配置为可选，而 `string` 将配置为“必需”。

下面的示例显示了一个具有必需和可选属性的实体类型，禁用了可为 `null` 的引用功能（默认值），并启用了该功能：

- [没有可为 null 的引用类型（默认值）](#)
- [具有可以为 null 的引用类型](#)

```
public class CustomerWithoutNullableReferenceTypes
{
    public int Id { get; set; }
    [Required] // Data annotations needed to configure as required
    public string FirstName { get; set; }
    [Required]
    public string LastName { get; set; } // Data annotations needed to configure as required
    public string MiddleName { get; set; } // Optional by convention
}
```

建议使用可以为 null 的引用类型，因为它会将 C# 代码中表示的可为 null 性传递到 EF Core 的模型和数据库，并免去使用熟知 API 或数据批注来两次表示相同的概念。

NOTE

在现有项目上启用可以为 null 的引用类型时要格外小心：现在配置为可选的引用类型属性现在将配置为“必需”，除非它们显式批注为可为 null。管理关系数据库架构时，这可能会导致生成更改数据库列的为 null 性的迁移。

有关可以为 null 的引用类型以及如何将其与 EF Core 一起使用的详细信息，[请参阅此功能的专用文档页](#)。

显式配置

可以按如下所示将“约定”可以为“可选”的属性配置为“必需”：

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    public int BlogId { get; set; }
    [Required]
    public string Url { get; set; }
}
```

Keys

2020/3/11 •

键充当每个实体实例的唯一标识符。EF 中的大多数实体都有一个键，此键映射到关系数据库中 **主键** 的概念（对于没有键的实体，请参阅[无键实体](#)）。实体可以有超过主键的其他键（有关详细信息，请参阅[备用键](#)）。

按照约定，将名为 `Id` 或 `<type name>Id` 的属性配置为实体的主键。

```
class Car
{
    public string Id { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}

class Truck
{
    public string TruckId { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

NOTE

[拥有的实体类型](#) 使用不同的规则来定义密钥。

可以将单个属性配置为实体的主键，如下所示：

- [数据注释](#)
- [熟知 API](#)

```
class Car
{
    [Key]
    public string LicensePlate { get; set; }

    public string Make { get; set; }
    public string Model { get; set; }
}
```

你还可以将多个属性配置为实体的键，这称为组合键。复合密钥只能使用熟知的 API 进行配置；约定将永远不会设置组合键，你不能使用数据批注来配置它。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasKey(c => new { c.State, c.LicensePlate });
}
```

主键名称

按照约定，使用名称 `PK_<type name>` 创建关系数据库主键。可以按如下所示配置 primary key 约束的名称：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasKey(b => b.BlogId)
        .HasName("PrimaryKey_BlogId");
}
```

键类型和值

虽然 EF Core 支持使用任何基元类型的属性作为主键，包括 `string`、`Guid`、`byte[]` 和其他类型，但并非所有数据库都支持将所有类型作为键。在某些情况下，可以自动将键值转换为支持的类型，否则应手动指定转换。

在将新实体添加到上下文时，键属性必须始终具有非默认值，但某些类型将由数据库生成。在这种情况下，当添加实体进行跟踪时，EF 会尝试生成一个临时值。在调用 `SaveChanges` 后，临时值将替换为数据库生成的值。

IMPORTANT

如果某个键属性的值由数据库生成，而在添加实体时指定了一个非默认值，则 EF 将假定该实体在数据库中已存在，并且将尝试对其进行更新而不是插入一个新的值。若要避免这种情况，请禁用值生成或了解[如何为生成的属性指定显式值](#)。

备用键

除了主键外，备用键还可用作每个实体实例的替代唯一标识符；它可用作关系的目标。使用关系数据库时，这将映射到备用键列上的唯一索引/约束和引用列的一个或多个外键约束的概念。

TIP

如果只是想要在列上强制唯一性，请定义唯一索引而不是备用键（请参阅[索引](#)）。在 EF 中，备用键为只读，并在唯一索引之上提供附加语义，因为它们可用作外键的目标。

系统通常会在需要时为你引入备用键，你无需手动配置它们。按照约定，当您标识的属性不是作为关系目标的主键时，将为您引入备用密钥。

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogUrl)
            .HasPrincipalKey(b => b.Url);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public string BlogUrl { get; set; }
    public Blog Blog { get; set; }
}

```

你还可以将单个属性配置为备用密钥：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => c.LicensePlate);
}

```

你还可以将多个属性配置为备用密钥(称为复合备用密钥)：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => new { c.State, c.LicensePlate });
}

```

最后，按照约定，为备用键引入的索引和约束将被命名为 `AK_<type name>_<property name>` (对于复合备用键 `<property name>` 变成以下划线分隔的属性名称列表)。您可以配置备用密钥的 index 和 unique 约束的名称：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Car>()
        .HasAlternateKey(c => c.LicensePlate)
        .HasName("AlternateKey_LicensePlate");
}

```

生成的值

2020/3/11 ·

值生成模式

有三个可用于属性的值生成模式：

- 无值生成
- 在添加时生成值
- 在添加或更新时生成值

无值生成

没有值生成意味着，需始终提供要保存到数据库的有效值。必须先将有效的值赋予新的实体，再将这些新的实体添加到上下文中。

在添加时生成值

在添加时生成值，意思是为新实体生成值。

根据所用数据库提供程序的不同，值可能会通过 EF 在客户端生成或者由数据库生成。如果由数据库生成值，则当你将实体添加到上下文时，EF 可能会赋予一个临时值。在 `SaveChanges()` 期间，此临时值将替换为数据库生成的值。

如果将一个实体添加到已经为属性赋予值的上下文，则 EF 会尝试插入该值而不是生成新值。如果某个属性未分配 CLR 默认值 (`null` 用于 `string`, `0` 用于 `int`, `Guid.Empty` `Guid` 等)，则该属性被视为已分配值。有关详细信息，请参阅[生成的属性的显式值](#)。

WARNING

如何为添加的实体生成值取决于所用数据库提供程序。数据库提供程序可能会为某些属性类型自动设置值的生成，但其他的属性类型可能要求你手动设置值的生成方式。

例如，使用 SQL Server 时，将自动为 `GUID` 属性（使用 SQL Server 顺序 GUID 算法）生成值。但是，如果您指定在添加时生成 `DateTime` 属性，则必须设置一个方法来生成值。执行此操作的一种方法是配置默认值 `GETDATE()`，请参阅[默认值](#)。

在添加或更新时生成值

在添加或更新时生成值，意味着在每次保存该记录（插入或更新）时生成新值。

与 `value generated on add` 一样，如果为新添加的实体实例的属性指定值，则将插入该值，而不是要生成的值。还可以在更新时设置显式值。有关详细信息，请参阅[生成的属性的显式值](#)。

WARNING

如何在添加和更新实体时生成值取决于所用数据库提供程序。数据库提供程序可能会为某些属性类型自动设置值的生成，但其他的属性类型会要求你手动设置值的生成方式。

例如，使用 SQL Server 时，将使用 `rowversion` 数据类型设置在添加或更新时生成的 `byte[]` 属性，并将其标记为并发标记，以便在数据库中生成值。但是，如果您指定在添加或更新时生成 `DateTime` 属性，则必须设置一个方法来生成值。实现此目的的一种方法是将默认值 `GETDATE()`（请参阅[默认值](#)）配置为新行生成值。然后即可使用数据库触发器在更新过程中生成值（如下面的示例触发器所示）。

```
CREATE TRIGGER [dbo].[Blogs_UPDATE] ON [dbo].[Blogs]
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF ((SELECT TRIGGER_NESTLEVEL()) > 1) RETURN;

    DECLARE @Id INT

    SELECT @Id = INSERTED.BlogId
    FROM INSERTED

    UPDATE dbo.Blogs
    SET LastUpdated = GETDATE()
    WHERE BlogId = @Id
END
```

在添加时生成值

按照约定，如果应用程序不提供值，则将 "short"、"int"、"long" 或 "Guid" 类型的非复合主键设置为为插入的实体生成值。您的数据库提供程序通常会负责必要的配置；例如，SQL Server 中的数字主键将自动设置为标识列。

可以将任何属性配置为为插入的实体生成其值，如下所示：

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public DateTime Inserted { get; set; }
}
```

WARNING

这只是让 EF 知道为已添加的实体生成值，并不保证 EF 会设置实际机制来生成值。有关更多详细信息，请参阅[add 部分生成的值](#)。

默认值

在关系数据库中，可以使用默认值来配置列；如果插入的行没有该列的值，将使用默认值。

可以在属性上配置默认值：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Rating)
        .HasDefaultValue(3);
}
```

还可以指定用于计算默认值的 SQL 片段：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Created)
        .HasDefaultValueSql("getdate()");
}
```

指定默认值会隐式将属性配置为在添加时生成的值。

在添加或更新时生成值

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime LastUpdated { get; set; }
}
```

WARNING

这只是让 EF 知道为添加或更新的实体生成值，并不保证 EF 会设置实际机制来生成值。有关更多详细信息，请参阅[add or update 部分上生成的值](#)。

计算列

在某些关系数据库上，列可以配置为在数据库中计算其值，通常具有引用其他列的表达式：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .Property(p => p.DisplayName)
        .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
}
```

NOTE

在某些情况下，每次提取列的值（有时称为虚拟列）时，都将计算该列的值，而在其他情况下，会在每次更新行时计算该列的值并存储（有时称为存储列或持久化列）。这不同于数据库提供程序。

无值生成

如果对属性的值生成进行了配置，则通常需要对其进行值生成。例如，如果你有一个 int 类型的主键，则它将被隐

式设置配置为在 add 时生成的值;可以通过以下方式禁用此操作:

- 数据注释
- 熟知 API

```
public class Blog
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

并发标记

2020/3/11 ·

NOTE

此页介绍如何配置并发标记。有关并发控制的工作原理的详细说明, 请参阅[处理并发冲突EF Core](#) 和[如何处理应用程序中的并发冲突的示例。](#)

配置为并发标记的属性用于实现乐观并发控制。

配置

- [数据注释](#)
- [熟知 API](#)

```
public class Person
{
    public int PersonId { get; set; }

    [ConcurrencyCheck]
    public string LastName { get; set; }

    public string FirstName { get; set; }
}
```

Timestamp/rowversion

Timestamp/rowversion 是一个属性, 在每次插入或更新行时, 数据库会自动为其生成新值。此属性也被视为并发标记, 这确保了在你查询行后, 如果正在更新的行发生了更改, 则会出现异常。确切的详细信息取决于所使用的数据库提供程序;对于 SQL Server, 通常使用`byte[]`属性, 该属性将设置为数据库中的`ROWVERSION`列。

可以按如下所示将属性配置为 timestamp/rowversion:

- [数据注释](#)
- [熟知 API](#)

```
public class Blog
{
    public int BlogId { get; set; }

    public string Url { get; set; }

    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

阴影属性

2020/3/11 •

影子属性是未在 .NET 实体类中定义但在 EF Core 模型中为该实体类型定义的属性。这些属性的值和状态纯粹在更改跟踪器中进行维护。当数据库中的数据不应在映射的实体类型上公开时，阴影属性非常有用。

外键阴影属性

影子属性最常用于外键属性，其中两个实体之间的关系由数据库中的外键值表示，但使用实体之间的导航属性在实体类型上管理关系各种。按照约定，当发现关系但在依赖实体类中找不到外键属性时，EF 会引入一个影子属性。

属性将命名为 `<navigation property name><principal key property name>`（指向主体实体的依赖实体上的导航用于命名）。如果主体键属性名称包含导航属性的名称，则该名称将只 `<principal key property name>`。如果依赖实体上没有导航属性，则会在其位置使用主体类型名称。

例如，下面的代码列表将导致向 `Post` 实体引入 `BlogId` 影子属性：

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    // Since there is no CLR property which holds the foreign
    // key for this relationship, a shadow property is created.
    public Blog Blog { get; set; }
}
```

配置阴影属性

你可以使用 "熟知 API" 配置阴影属性。在您调用了 `Property` 的字符串重载后，您可以将对其他属性的任何配置调用链接在一起。在下面的示例中，由于 `Blog` 没有名为 `LastUpdated` 的 CLR 属性，因此将创建一个影子属性：

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property<DateTime>("LastUpdated");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}
```

如果提供给 `Property` 方法的名称与现有属性(在实体类中定义)的名称相匹配，则代码将配置该现有属性，而不是引入新的阴影属性。

访问阴影属性

可以通过 `ChangeTracker` API 获取和更改影子属性值：

```
context.Entry(myBlog).Property("LastUpdated").CurrentValue = DateTime.Now;
```

可以通过 `EF.Property` 静态方法在 LINQ 查询中引用阴影属性：

```
var blogs = context.Blogs
    .OrderBy(b => EF.Property<DateTime>(b, "LastUpdated"));
```

关系

2020/3/16 •

关系定义两个实体之间的关系。在关系数据库中，这由外键约束表示。

NOTE

本文中的大多数示例都使用一对多关系来演示概念。有关一对一关系和多对多关系的示例，请参阅文章末尾的[其他关系模式部分](#)。

术语定义

有许多术语用于描述关系

- **相关实体**: 这是包含外键属性的实体。有时称为关系的 "子级"。
- **主体实体**: 这是包含主/备用键属性的实体。有时称为关系的 "父项"。
- **主体密钥**: 唯一标识主体实体的属性。这可能是主键或备用密钥。
- **外键**: 用于存储相关实体的主体键值的依赖实体中的属性。
- **导航属性**: 在主体和/或从属实体上定义的属性，该属性引用相关实体。
 - **集合导航属性**: 一个导航属性，其中包含对多个相关实体的引用。
 - **引用导航属性**: 保存对单个相关实体的引用的导航属性。
 - **反向导航属性**: 讨论特定导航属性时，此术语是指关系另一端的导航属性。
- **自引用关系**: 依赖关系和主体实体类型相同的关系。

下面的代码显示 `Blog` 与 `Post` 之间的一对多关系。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

- `Post` 是依赖实体
- `Blog` 是主体实体
- `Blog.BlogId` 是主体键(在本例中为主键而不是备用密钥)

- `Post.BlogId` 为外键
- `Post.Blog` 是一个引用导航属性
- `Blog.Posts` 是集合导航属性
- `Post.Blog` 是 `Blog.Posts` 的反向导航属性(反之亦然)

约定

默认情况下，当在某个类型上发现导航属性时，将创建一个关系。如果属性指向的类型不能由当前的数据库提供程序映射为标量类型，则该属性视为一个导航属性。

NOTE

按约定发现的关系将始终以主体实体的主键为目标。若要以备用密钥为目标，则必须使用熟知的 API 执行其他配置。

完全定义的关系

关系最常见的模式是在关系两端定义导航属性，在依赖实体类中定义外键属性。

- 如果在两个类型之间找到一对导航属性，则这些属性将配置为同一关系的反向导航属性。
- 如果依赖实体包含名称与其中一种模式相匹配的属性，则该属性将被配置为外键：

- <navigation property name><principal key property name>
- <navigation property name>Id
- <principal entity name><principal key property name>
- <principal entity name>Id

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

在此示例中，突出显示的属性将用于配置关系。

NOTE

如果属性为主键，或者为与主体键不兼容的类型，则不会将其配置为外键。

NOTE

在 EF Core 3.0 之前，名为与主体键属性完全相同的属性[也与外键匹配](#)

无外键属性

尽管建议在依赖实体类中定义外键属性，但这并不是必需的。如果未找到外键属性，则会引入名称为

<navigation property name><principal key property name> 或 <principal entity name><principal key property name> 的阴影外键属性（如果依赖类型上没有导航）。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

在此示例中，隐藏外键是 `BlogId` 的，因为预先计算导航名称将是冗余的。

NOTE

如果已存在具有相同名称的属性，则会以数字作为后缀的阴影属性名称。

单个导航属性

只包含一个导航属性（无反向导航，没有外键属性）就足以具有约定定义的关系。还可以有一个导航属性和一个外键属性。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
```

限制

如果有多个在两种类型之间定义的导航属性（即，不仅仅是一对相互指向的导航属性），则导航属性表示的关系是不明确的。你将需要手动对其进行配置以解决歧义。

级联删除

按照约定，级联删除将对所需的关系和 `ClientSetNull` 设置为 `cascade`，以实现可选关系。`Cascade` 表示也会删除依赖实体。`ClientSetNull` 表示未加载到内存中的依赖实体将保持不变，必须手动删除，或将其更新为指向有效的主体实体。对于加载到内存中的实体，EF Core 将尝试将外键属性设置为 `null`。

请参阅 [required](#) 和 [optional](#) 关系部分，了解必需和可选关系之间的差异。

有关不同的删除行为和约定使用的默认值的详细信息，请参阅[级联删除](#)。

手动配置

- 熟知 API
- 数据批注

若要在熟知的 API 中配置关系，请首先标识构成关系的导航属性。`HasOne` 或 `HasMany` 标识正在开始配置的实体类型上的导航属性。然后，将调用链接到 `WithOne` 或 `WithMany` 来标识反向导航。`HasOne / WithOne` 用于引用导航属性，`HasMany / 用于集合导航属性。WithMany`

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}
```

单个导航属性

如果只有一个导航属性，则 `WithOne` 和 `WithMany` 有无参数重载。这表示在概念上，关系的另一端有一个引用或集合，但实体类中不包含导航属性。

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasMany(b => b.Posts)
            .WithOne();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
}
```

外键

- [熟知 API \(简单密钥\)](#)
- [熟知 API \(组合键\)](#)
- [数据批注\(简单键\)](#)

您可以使用熟知的 API 来配置应用作给定关系的外键属性：

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey(p => p.BlogForeignKey);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
```

影子外键

您可以使用 `HasForeignKey(...)` 的字符串重载将影子属性配置为外键(有关详细信息, 请参阅[影子属性](#))。建议先将影子属性显式添加到模型, 然后再将其用作外键(如下所示)。

```

class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // Add the shadow property to the model
        modelBuilder.Entity<Post>()
            .Property<int>("BlogForeignKey");

        // Use the shadow property as a foreign key
        modelBuilder.Entity<Post>()
            .HasOne(p => p.Blog)
            .WithMany(b => b.Posts)
            .HasForeignKey("BlogForeignKey");
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

```

Foreign key 约束名称

按照约定，以关系数据库为目标时，外键约束将命名 `FK_`。对于复合外键，成为外键属性名称的以下划线分隔的列表。

你还可以配置约束名称，如下所示：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .HasForeignKey(p => p.BlogId)
        .HasConstraintName("ForeignKey_Post_Blog");
}

```

无导航属性

不一定需要提供导航属性。您可以直接在关系的一端提供外键。

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Post>()
            .HasOne<Blog>()
            .WithMany()
            .HasForeignKey(p => p.BlogId);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
}
```

主体密钥

如果你希望外键引用主键之外的属性，则可以使用熟知的 API 来配置关系的主体键属性。配置为主体密钥的属性将自动设置为[备用密钥](#)。

- [简单键](#)
- [组合键](#)

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => s.CarLicensePlate)
            .HasPrincipalKey(c => c.LicensePlate);
    }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public List<RecordOfSale> SaleHistory { get; set; }
}

public class RecordOfSale
{
    public int RecordOfSaleId { get; set; }
    public DateTime DateSold { get; set; }
    public decimal Price { get; set; }

    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}

```

必需和可选的关系

您可以使用熟知的 API 来配置关系是必需的还是可选的。最终，这会控制外键属性是必需的还是可选的。当使用阴影状态外键时，这非常有用。如果实体类中具有外键属性，则关系的 requiredness 取决于外键属性是必需还是可选（有关详细信息，请参阅[必需和可选属性](#)）。

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .IsRequired();
}

```

NOTE

调用 `IsRequired(false)` 还会使外键属性为可选，除非已对其进行配置。

级联删除

您可以使用熟知的 API 显式配置给定关系的级联删除行为。

有关每个选项的详细讨论，请参阅[级联删除](#)。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasOne(p => p.Blog)
        .WithMany(b => b.Posts)
        .OnDelete(DeleteBehavior.Cascade);
}
```

其他关系模式

一对一

一对多关系在两侧都有一个引用导航属性。它们遵循与一对多关系相同的约定，但在外键属性上引入了唯一索引，以确保只有一个依赖项与每个主体相关。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

NOTE

EF 会根据其检测外键属性的能力，选择其中一个实体作为依赖项。如果选择了错误的实体作为依赖项，则可以使用熟知的 API 来更正此问题。

使用熟知 API 配置关系时，可使用 `HasOne` 和 `WithOne` 方法。

配置外键时，需要指定依赖实体类型-请注意以下列表中 `HasForeignKey` 提供的泛型参数。在一对多关系中，可以清楚地表明具有引用导航的实体是依赖项，并且具有集合的实体是主体。但这并不是一对一的关系，因此需要显式定义它。

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<BlogImage> BlogImages { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasOne(b => b.BlogImage)
            .WithOne(i => i.Blog)
            .HasForeignKey<BlogImage>(b => b.BlogForeignKey);
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogForeignKey { get; set; }
    public Blog Blog { get; set; }
}
```

多对多

目前尚不支持多对多关系，没有实体类来表示联接表。但是，您可以通过包含联接表的实体类并映射两个不同的
一对多关系，来表示多对多关系。

```
class MyContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Tag> Tags { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<PostTag>()
            .HasKey(t => new { t.PostId, t.TagId });

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Post)
            .WithMany(p => p.PostTags)
            .HasForeignKey(pt => pt.PostId);

        modelBuilder.Entity<PostTag>()
            .HasOne(pt => pt.Tag)
            .WithMany(t => t.PostTags)
            .HasForeignKey(pt => pt.TagId);
    }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public List<PostTag> PostTags { get; set; }
}

public class PostTag
{
    public int PostId { get; set; }
    public Post Post { get; set; }

    public string TagId { get; set; }
    public Tag Tag { get; set; }
}
```

索引

2020/3/11 ·

索引是跨多个数据存储区的常见概念。尽管它们在数据存储中的实现可能会有所不同，但也可用于基于列（或一组列）更高效地进行查找。

不能使用数据批注创建索引。您可以使用“熟知 API”按如下方式为单个列指定索引：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url);
}
```

您还可以为多个列指定索引：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>()
        .HasIndex(p => new { p.FirstName, p.LastName });
}
```

NOTE

按照约定，将在用作外键的每个属性（或一组属性）中创建索引。

EF Core 每个不同的属性集仅支持一个索引。如果使用“熟知 API”来配置已定义索引的属性集的索引（按照约定或以前的配置），则会更改该索引的定义。如果要进一步配置由约定创建的索引，则此操作非常有用。

索引唯一性

默认情况下，索引不唯一：允许多行具有与索引的列集相同的值。可以使索引唯一，如下所示：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .IsUnique();
}
```

尝试为索引的列集插入多个具有相同值的实体将导致引发异常。

索引名称

按照约定，在关系数据库中创建的索引将命名为 `IX_<type name>_<property name>`。对于复合索引，`<property name>` 变成以下划线分隔的属性名称列表。

您可以使用“熟知 API”设置在数据库中创建的索引的名称：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .HasName("Index_Url");
}
```

索引筛选器

某些关系数据库允许您指定筛选索引或部分索引。这使您可以只为列的值的一个子集编制索引，从而减少索引的大小并改善性能和磁盘空间的使用情况。有关 SQL Server 筛选索引的详细信息，[请参阅文档](#)。

您可以使用熟知的 API 来指定索引的筛选器，作为 SQL 表达式提供：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .HasFilter("[Url] IS NOT NULL");
}
```

当使用 SQL Server 提供程序 EF 时，将为唯一索引中包含的所有可以为 null 的列添加 'IS NOT NULL' 筛选器。若要重写此约定，可以提供 `null` 值。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasIndex(b => b.Url)
        .IsUnique()
        .HasFilter(null);
}
```

包含列

某些关系数据库允许配置一组列，这些列包含在索引中，但不是其 "键" 的一部分。当查询中的所有列都作为键列或非键列包含在索引中时，这可以显著提高查询性能，因为表本身无需访问。有关 SQL Server 包含列的详细信息，[请参阅文档](#)。

在下面的示例中，`Url` 列是索引键的一部分，因此对该列的任何查询筛选都可以使用索引。但此外，仅访问 `Title` 和 `PublishedOn` 列的查询将不需要访问表，并且将更有效地运行：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .HasIndex(p => p.Url)
        .IncludeProperties(p => new
        {
            p.Title,
            p.PublishedOn
        });
}
```

继承

2020/3/11 •

EF 可以将 .NET 类型层次结构映射到数据库。这使你可以像平常一样使用基类型和派生类型在代码中编写 .NET 实体，并让 EF 无缝创建适当的数据库架构、发出查询等。类型层次结构的映射方式的实际详细信息与提供程序相关；本页介绍关系数据库上下文中的继承支持。

目前，EF Core 仅支持每个层次结构一个表(TPH)模式。TPH 使用单个表来存储层次结构中所有类型的数据，而鉴别器列用于标识每行所表示的类型。

NOTE

EF Core 尚不支持 EF6 支持每种类型一个表(TPT)和每个具体的表类型(TPC)。TPT 是为 EF Core 5.0 计划的主要功能。

实体类型层次结构映射

按照约定，EF 仅会在模型中显式包括两个或更多个继承类型时设置继承。EF 不会自动扫描未在模型中包括的基类型或派生类型。

可以通过为继承层次结构中的每个类型公开 DbSet，在模型中包括类型：

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<RssBlog> RssBlogs { get; set; }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
}

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

此模型映射到下面的数据库架构(请注意隐式创建的鉴别器列，该列标识每个行中存储的博客类型)：

Results			
	BlogId	Discriminator	Url
1	1	Blog	http://blogs.msdn.com/dotnet
2	2	RssBlog	http://blogs.msdn.com/adonet

NOTE

使用 TPH 映射时，数据库列会根据需要自动进行为 null。例如，RssUrl 列可以为 null，因为常规博客实例不具有该属性。

如果不想公开层次结构中一个或多个实体的 DbSet，还可以使用熟知的 API 来确保它们包含在模型中。

TIP

如果不依赖约定，可以使用 `.HasBaseType` 显式指定基类型。你还可以使用 `.HasBaseType((Type)null)` 从层次结构中删除实体类型。

鉴别器配置

您可以配置鉴别器列的名称和类型以及用于标识层次结构中的每种类型的值：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasDiscriminator<string>("blog_type")
        .HasValue<Blog>("blog_base")
        .HasValue<RssBlog>("blog_rss");
}
```

在上述示例中，EF 在层次结构的基实体上将鉴别器隐式添加为 [影子属性](#)。此属性可以配置为类似于任何其他属性：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property("Discriminator")
        .HasMaxLength(200);
}
```

最后，鉴别器还可以映射到实体中的常规 .NET 属性：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .HasDiscriminator(b => b.BlogType);

    modelBuilder.Entity<Blog>()
        .Property(e => e.BlogType)
        .HasMaxLength(200)
        .HasColumnName("blog_type");
}
```

共享列

默认情况下，当层次结构中的两个同级实体类型具有相同的属性时，它们将映射到两个单独的列。但是，如果它们的类型相同，则可以映射到相同的数据库列：

```
public class MyContext : DbContext
{
    public DbSet<BlogBase> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasColumnName("Url");

        modelBuilder.Entity<RssBlog>()
            .Property(b => b.Url)
            .HasColumnName("Url");
    }
}

public abstract class BlogBase
{
    public int BlogId { get; set; }
}

public class Blog : BlogBase
{
    public string Url { get; set; }
}

public class RssBlog : BlogBase
{
    public string Url { get; set; }
}
```

序列

2020/3/11 •

NOTE

序列是通常仅由关系数据库支持的功能。如果使用的是非关系数据库(如 Cosmos)，请查看数据库文档以生成唯一值。

序列将在数据库中生成唯一的顺序数值。序列不与特定表相关联，并且可以将多个表设置为从同一序列中绘制值。

基本用法

您可以在模型中设置一个序列，然后使用它来生成属性值：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasSequence<int>("OrderNumbers");

    modelBuilder.Entity<Order>()
        .Property(o => o.OrderNo)
        .HasDefaultValueSql("NEXT VALUE FOR shared.OrderNumbers");
}
```

请注意，用于从序列生成值的特定 SQL 是特定于数据库的；上面的示例可在 SQL Server 上运行，但在其他数据库中将失败。有关详细信息，请参阅特定数据库的文档。

配置序列设置

你还可以配置序列的其他方面，例如其架构、起始值、增量等：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
        .StartsAt(1000)
        .IncrementsBy(5);
}
```

支持字段

2020/3/11 •

支持字段允许 EF 读取和/或写入字段，而不是属性。当使用类中的封装来限制和/或通过应用程序代码对数据访问进行限制时，这可能很有用，但在不使用这些限制/增强功能的情况下，应从数据库中读取和/或写入值。

基本配置

按照约定，将发现以下字段作为给定属性的支持字段（按优先级顺序列出）。

- `_<camel-cased property name>`
- `_<property name>`
- `m_<camel-cased property name>`
- `m_<property name>`

在下面的示例中，`Url` 属性配置为具有与其支持字段 `_url`：

```
public class Blog
{
    private string _url;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```

请注意，仅为模型中包含的属性发现支持字段。有关模型中包含哪些属性的详细信息，请参阅[包括 & 排除属性](#)。

还可以显式配置支持字段，例如，如果字段名称与上述约定不对应：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .HasField("_validatedUrl");
}
```

字段和属性访问

默认情况下，EF 将始终读取并写入到支持字段-假设已正确配置了一个字段，并且永远不会使用属性。但是，EF 还支持其他访问模式。例如，下面的示例指示 EF 仅在具体化时写入支持字段，并在所有其他情况下使用属性：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property(b => b.Url)
        .HasField("_validatedUrl")
        .UsePropertyAccessMode(PropertyAccessMode.PreferFieldDuringConstruction);
}
```

有关完整的支持选项集, 请参阅[PropertyAccessMode 枚举](#)。

NOTE

使用 EF Core 3.0, 默认属性访问模式从 `PreferFieldDuringConstruction` 改为 `PreferField`。

仅限字段的属性

您还可以在您的模型中创建一个概念属性, 该属性在实体类中不具有相应的 CLR 属性, 而是使用字段来存储实体中的数据。这不同于[阴影属性](#), 其中的数据存储在更改跟踪器中, 而不是存储在实体的 CLR 类型中。仅字段属性在实体类使用方法而不是属性来获取/设置值时使用, 或者在字段不应在域模型中公开(例如主键)的情况下使用。

可以通过在 `Property(...)` API 中提供名称来配置仅限字段的属性:

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property("validatedUrl");
    }
}

public class Blog
{
    private string _validatedUrl;

    public int BlogId { get; set; }

    public string GetUrl()
    {
        return _validatedUrl;
    }

    public void SetUrl(string url)
    {
        using (var client = new HttpClient())
        {
            var response = client.GetAsync(url).Result;
            response.EnsureSuccessStatusCode();
        }

        _validatedUrl = url;
    }
}
```

EF 将尝试查找具有给定名称的 CLR 属性, 如果找不到属性, 则尝试查找一个字段。如果属性和字段均未找到, 则将改为设置影子属性。

您可能需要从 LINQ 查询中引用仅限字段的属性, 但此类字段通常是私有的。可以在 LINQ 查询中使用

EF.Property(...) 方法来引用字段:

```
var blogs = db.blogs.OrderBy(b => EF.Property<string>(b, "_validatedUrl"));
```

值转换

2020/3/11 •

NOTE

此功能是 EF Core 2.1 中的新增功能。

值转换器允许在读取或写入数据库时转换属性值。此转换可以是同一类型的另一个值(例如, 加密字符串)或从一种类型的值转换为另一种类型的值(例如, 在数据库中将枚举值与字符串相互转换)。

基础

值转换器根据 `ModelClrType` 和 `ProviderClrType` 来指定。模型类型是实体类型中的属性的 .NET 类型。提供程序类型是数据库提供程序理解的 .NET 类型。例如, 若要将枚举作为字符串保存在数据库中, 模型类型是枚举的类型, 提供程序类型为 `String`。这两种类型可以相同。

使用两个 `Func` 表达式树来定义转换:一个从 `ModelClrType` 到 `ProviderClrType`, 另一个从 `ProviderClrType` 到 `ModelClrType`。使用表达式树, 以便可以将它们编译到数据库访问代码中以便进行有效的转换。对于复杂转换, 表达式树可能是对执行转换的方法的简单调用。

配置值转换器

值转换是在 `DbContext` 的 `OnModelCreating` 中的属性上定义的。例如, 假设枚举和实体类型定义为:

```
public class Rider
{
    public int Id { get; set; }
    public EquineBeast Mount { get; set; }
}

public enum EquineBeast
{
    Donkey,
    Mule,
    Horse,
    Unicorn
}
```

然后, 可以在 `OnModelCreating` 中定义转换, 以将枚举值存储为数据库中的字符串(例如, "Donkey"、"Mule" 和 ...) :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(
            v => v.ToString(),
            v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));
}
```

NOTE

`null` 值永远不会传递到值转换器。这使得转换的实现变得更简单，并使其能够在可以为 `null` 和不可为 `null` 的属性之间共享。

ValueConverter 类

如上所述调用 `HasConversion` 将创建一个 `ValueConverter` 实例，并在属性上对其进行设置。可以改为显式创建 `ValueConverter`。例如：

```
var converter = new ValueConverter<EquineBeast, string>(
    v => v.ToString(),
    v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

当多个属性使用相同的转换时，这可能很有用。

NOTE

当前无法在一个位置指定给定类型的每个属性都必须使用相同的值转换器。将来的版本将考虑此功能。

内置转换器

EF Core 附带一组预定义的 `ValueConverter` 类，这些类在 `Microsoft.EntityFrameworkCore.Storage.ValueConversion` 命名空间中找到。这些位置包括：

- `BoolToZeroOneConverter` -布尔值到零和一个
- `BoolToStringConverter` -布尔值到字符串(如 "Y" 和 "N")
- `BoolToTwoValuesConverter` -布尔值为任意两个值
- `BytesToStringConverter` 字节数组到 Base64 编码的字符串
- `CastingConverter` -只需要类型强制转换的转换
- `CharToStringConverter` -Char 到单字符字符串
- `DateTimeOffsetToBinaryConverter` -DateTimeOffset 到二进制编码的64位值
- `DateTimeOffsetToBytesConverter` -DateTimeOffset 到字节数组
- `DateTimeOffsetToStringConverter` -DateTimeOffset 到字符串
- `DateTimeToBinaryConverter` -DateTime 到64位值(包括 Datetimekind.utc)
- `DateTimeToStringConverter` -DateTime 到 string
- `DateTimeToTicksConverter` -DateTime 到计时周期
- `EnumToNumberConverter` -枚举到基础数字
- `EnumToStringConverter` 枚举到字符串
- `GuidToBytesConverter` -Guid 到字节数组
- `GuidToStringConverter` -Guid 到字符串
- `NumberToBytesConverter` -任何数值到字节数组
- `NumberToStringConverter` -字符串的任何数值
- `StringToBytesConverter` 字符串到 UTF8 字节

- `TimeSpanToStringConverter` -`TimeSpan` 到字符串
- `TimeSpanToTicksConverter` -`TimeSpan` 到计时周期

请注意，`EnumToStringConverter` 包含在此列表中。这意味着无需显式指定转换，如上所示。相反，只需使用内置转换器：

```
var converter = new EnumToStringConverter<EquineBeast>();

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

请注意，所有内置的转换器都是无状态的，因此，多个属性可以安全地共享单个实例。

预定义的转换

对于内置转换器存在的常见转换，无需显式指定转换器。相反，只需配置应使用的提供程序类型，EF 会自动使用适当的内置转换器。枚举到字符串的转换用作上面的示例，但如果配置了提供程序类型，则 EF 实际上会自动执行此操作：

```
modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion<string>();
```

可以通过显式指定列类型来实现相同的目的。例如，如果定义了实体类型，如下所示：

```
public class Rider
{
    public int Id { get; set; }

    [Column(TypeName = "nvarchar(24)")]
    public EquineBeast Mount { get; set; }
}
```

然后，枚举值将以字符串形式保存在数据库中，而不会在 `OnModelCreating` 中进行任何进一步的配置。

限制

值转换系统存在一些已知的当前限制：

- 如上所述，不能转换 `null`。
- 目前没有办法将一个属性转换为多个列，反之亦然。
- 使用值转换可能会影响 EF Core 将表达式转换为 SQL 的能力。这种情况下会记录警告。将来的版本将考虑删除这些限制。

数据种子设定

2020/3/11 ·

数据种子是用初始数据集填充数据库的过程。

可以通过多种方式在 EF Core 中完成此操作：

- 模型种子数据
- 手动迁移自定义
- 自定义初始化逻辑

模型种子数据

NOTE

此功能是 EF Core 2.1 中的新增功能。

与在 EF6 中不同，在 EF Core 中，种子设定数据可以作为模型配置的一部分与实体类型相关联。然后 EF Core [迁移](#) 可以自动计算在将数据库升级到新版本的模型时需要应用的插入、更新或删除操作。

NOTE

迁移仅在确定应该执行哪种操作以使种子数据进入所需状态时才考虑模型更改。因此，在迁移外部执行的数据更改可能会丢失或导致错误。

例如，这将为 `OnModelCreating` 中的 `Blog` 配置种子数据：

```
modelBuilder.Entity<Blog>().HasData(new Blog { BlogId = 1, Url = "http://sample.com" });
```

若要添加具有关系的实体，需要指定外键值：

```
modelBuilder.Entity<Post>().HasData(  
    new Post() { BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1" });
```

如果实体类型具有隐藏状态的任何属性，则可以使用匿名类提供值：

```
modelBuilder.Entity<Post>().HasData(  
    new { BlogId = 1, PostId = 2, Title = "Second post", Content = "Test 2" });
```

拥有的实体类型可以采用类似的方式进行种子设定：

```
modelBuilder.Entity<Post>().OwnsOne(p => p.AuthorName).HasData(  
    new { PostId = 1, First = "Andriy", Last = "Svyryd" },  
    new { PostId = 2, First = "Diego", Last = "Vega" } );
```

有关更多上下文，请参阅[完整的示例项目](#)。

将数据添加到模型后，应使用[迁移](#)来应用更改。

TIP

如果需要在自动部署中应用迁移，可以创建一个可在执行前预览的SQL 脚本。

或者，可以使用 `context.Database.EnsureCreated()` 创建包含种子数据的新数据库，例如测试数据库，或使用内存中提供程序或任何非关系数据库时。请注意，如果数据库已存在，`EnsureCreated()` 将不会更新数据库中的架构或种子数据。对于关系数据库，如果您计划使用迁移，则不应调用 `EnsureCreated()`。

模型种子数据的限制

此类型的种子数据由迁移管理，需要在不连接到数据库的情况下生成用于更新数据库中已存在的数据的脚本。这会施加一些限制：

- 主键值需要指定，即使它通常由数据库生成。它用于检测迁移间的数据更改。
- 如果以任何方式更改了主键，则将删除以前的种子数据。

因此，此功能最适用于不需要在迁移外更改的静态数据，并且不依赖于数据库中的任何其他内容（例如，邮政编码）。

如果你的方案包括以下任何一种情况，则建议使用上一部分中所述的自定义初始化逻辑：

- 用于测试的临时数据
- 依赖于数据库状态的数据
- 需要由数据库生成的键值的数据，包括使用替代密钥作为标识的实体
- 需要自定义转换的数据（不由值转换处理），如某些密码哈希
- 需要调用外部 API 的数据，例如 ASP.NET Core 标识角色和用户创建

手动迁移自定义

添加迁移时，对使用 `HasData` 指定的数据所做的更改将转换为对 `InsertData()`、`UpdateData()` 和 `DeleteData()` 的调用。解决 `HasData` 的某些限制的一种方法是手动将这些调用或[自定义操作](#)添加到迁移。

```
migrationBuilder.InsertData(
    table: "Blogs",
    columns: new[] { "Url" },
    values: new object[] { "http://generated.com" });
```

自定义初始化逻辑

执行数据种子设定的一种简单而有效的方法是在主应用程序逻辑开始执行之前使用 `DbContext.SaveChanges()`。

```
using (var context = new DataSeedingContext())
{
    context.Database.EnsureCreated();

    var testBlog = context.Blogs.FirstOrDefault(b => b.Url == "http://test.com");
    if (testBlog == null)
    {
        context.Blogs.Add(new Blog { Url = "http://test.com" });
    }
    context.SaveChanges();
}
```

WARNING

种子设定代码不应是正常应用执行的一部分，因为这可能会导致多个实例运行时出现并发性问题，并且还要求应用有权修改数据库架构。

根据部署的约束，可以通过不同的方式执行初始化代码：

- 在本地运行初始化应用程序
- 将初始化应用与主应用一起部署，调用初始化例程，禁用或删除初始化应用。

通常可以使用[发布配置文件](#)自动完成此配置。

具有构造函数的实体类型

2020/3/11 •

NOTE

此功能是 EF Core 2.1 中的新增功能。

从开始 EF Core 2.1，它则现在可以定义参数的构造函数，并让 EF Core 创建实体的实例时调用此构造函数。构造函数参数可以绑定到映射的属性或各种类型的服务，以促进延迟加载等行为。

NOTE

截至 EF Core 2.1，按照约定将为所有构造函数绑定。计划在将来的版本中配置要使用的特定构造函数。

绑定到映射的属性

请考虑典型的博客/公告模型：

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}
```

当 EF Core 创建这些类型的实例时，如的结果的查询，它将首先调用默认的无参数构造函数，然后设置每个属性的值从数据库。但是，如果 EF Core 查找的参数化构造函数参数名称和类型相匹配的映射属性，则它将改为调用这些属性具有值的参数化构造函数，然后将未显式设置每个属性。例如：

```

public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; set; }

    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; set; }

    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}

```

需要注意的事项：

- 并非所有属性都需要具有构造函数参数。例如，由任何构造函数参数，因此 EF Core 将以正常方式调用的构造函数后设置未设置 Post.Content 属性。
- 参数类型和名称必须与属性类型和名称相匹配，但在参数采用 camel 大小写格式时，属性可以采用 Pascal 大小写形式。
- 无法设置（如博客或更高版本的文章）的导航属性，EF Core 使用构造函数。
- 构造函数可以是公共的，也可以是私有的，或者具有任何其他可访问性。不过，延迟加载代理要求构造函数可以从继承代理类访问。通常，这意味着将其设为公共或受保护。

只读属性

通过构造函数设置属性后，就可以使某些属性成为只读的。EF Core 支持此功能，但有一些需要注意的事项：

- 不会按约定映射没有 setter 的属性。（这样做常常会映射不应映射的属性，例如计算属性。）
- 使用自动生成的键值需要键属性，该属性是读写的，因为在插入新实体时密钥生成器需要设置密钥值。

避免这种情况的简单方法是使用专用资源库。例如：

```
public class Blog
{
    public Blog(int id, string name, string author)
    {
        Id = id;
        Name = name;
        Author = author;
    }

    public int Id { get; private set; }

    public string Name { get; private set; }
    public string Author { get; private set; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public Post(int id, string title, DateTime postedOn)
    {
        Id = id;
        Title = title;
        PostedOn = postedOn;
    }

    public int Id { get; private set; }

    public string Title { get; private set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; private set; }

    public Blog Blog { get; set; }
}
```

EF Core 看到为读写模式，这意味着，像以前那样映射所有属性并密钥可能仍会由存储生成的具有专用 setter 的属性。

使用专用资源库的一种替代方法是使属性真正为只读，并在 OnModelCreating 中添加更多显式映射。同样，可以完全删除某些属性并仅替换为字段。例如，请考虑以下实体类型：

```

public class Blog
{
    private int _id;

    public Blog(string name, string author)
    {
        Name = name;
        Author = author;
    }

    public string Name { get; }
    public string Author { get; }

    public ICollection<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    private int _id;

    public Post(string title, DateTime postedOn)
    {
        Title = title;
        PostedOn = postedOn;
    }

    public string Title { get; }
    public string Content { get; set; }
    public DateTime PostedOn { get; }

    public Blog Blog { get; set; }
}

```

OnModelCreating 中的此配置：

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Author);
            b.Property(e => e.Name);
        });

    modelBuilder.Entity<Post>(
        b =>
        {
            b.HasKey("_id");
            b.Property(e => e.Title);
            b.Property(e => e.PostedOn);
        });
}

```

注意事项：

- "属性" 键现在是字段。它不是 `readonly` 字段，因此可以使用存储生成的键。
- 其他属性是仅在构造函数中设置的只读属性。
- 如果主键值只是由 EF 设置或从数据库中读取，则无需将其包含在构造函数中。这会使 "属性" 键作为一个简单字段，并清楚地指出不应在创建新的博客或文章时显式设置。

NOTE

此代码将导致编译器警告 "169", 指示该字段从未使用过。由于在现实中 EF Core extralinguistic 的方式使用该字段, 这可予以忽视。

注入服务

EF Core 还可以将"服务"注入到实体类型的构造函数。例如, 可以注入以下内容:

- `DbContext` -当前上下文实例, 也可以类型化为派生的 `DbContext` 类型
- `ILazyLoader` -延迟加载服务-有关更多详细信息, 请参阅[延迟加载文档](#)
- `Action<object, string>` -延迟加载委托--有关更多详细信息, 请参阅[延迟加载文档](#)
- `IEntityType` -与此实体类型关联的 EF Core 元数据

NOTE

截至 EF Core 2.1, 可插入仅通过 EF Core 已知的服务。将来的版本会考虑对注入应用程序服务的支持。

例如, 注入的 `DbContext` 可用于有选择地访问数据库, 以获取相关实体的相关信息, 而无需将它们全部加载。在下面的示例中, 这用于在不加载帖子的情况下获取博客中的帖子数:

```
public class Blog
{
    public Blog()
    {
    }

    private Blog(BloggingContext context)
    {
        Context = context;
    }

    private BloggingContext Context { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }
    public string Author { get; set; }

    public ICollection<Post> Posts { get; set; }

    public int PostsCount
        => Posts?.Count
        ?? Context?.Set<Post>().Count(p => Id == EF.Property<int?>(p, "BlogId"))
        ?? 0;
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime PostedOn { get; set; }

    public Blog Blog { get; set; }
}
```

请注意以下几点:

- 构造函数是专用容器, 因为它只能由 EF Core, 并且没有用于常规用途的另一个公共构造函数。

- 使用注入的服务(即上下文)的代码可防御 `null` 处理 EF Core 不创建实例的情况。
- 因为服务存储在读/写属性中, 所以当将实体附加到新的上下文实例时, 它将被重置。

WARNING

因为它将直接与 EF Core 的实体类型, 将注入如下 `DbContext` 通常被视为反模式。使用此类服务注入之前, 请仔细考虑所有选项。

表拆分

2020/3/11 •

EF Core 允许将两个或多个实体映射到单个行。这称为 "表拆分" 或 "表共享"。

配置

若要使用表拆分，则需要将多个实体类型映射到同一个表中，将主键映射到相同的列，并且在同一个实体类型的主键与另一个实体类型的主键之间配置至少一个关系。

表拆分的常见方案是只使用表中的部分列，以获得更好的性能或封装。

在此示例中 `Order` 表示 `DetailedOrder` 的子集。

```
public class Order
{
    public int Id { get; set; }
    public OrderStatus? Status { get; set; }
    public DetailedOrder DetailedOrder { get; set; }
}
```

```
public class DetailedOrder
{
    public int Id { get; set; }
    public OrderStatus? Status { get; set; }
    public string BillingAddress { get; set; }
    public string ShippingAddress { get; set; }
    public byte[] Version { get; set; }
}
```

除了所需的配置之外，我们还 `Property(o => o.Status).HasColumnName("Status")` 将 `DetailedOrder.Status` 映射到与 `Order.Status` 相同的列。

```
modelBuilder.Entity<DetailedOrder>(dob =>
{
    dob.ToTable("Orders");
    dob.Property(o => o.Status).HasColumnName("Status");
});

modelBuilder.Entity<Order>(ob =>
{
    ob.ToTable("Orders");
    ob.Property(o => o.Status).HasColumnName("Status");
    obhasOne(o => o.DetailedOrder).WithOne()
        .HasForeignKey<DetailedOrder>(o => o.Id);
});
```

TIP

有关更多上下文，请参阅[完整的示例项目](#)。

用法

使用表拆分来保存和查询实体的方式与其他实体相同：

```
using (var context = new TableSplittingContext())
{
    context.Database.EnsureDeleted();
    context.Database.EnsureCreated();

    context.Add(new Order
    {
        Status = OrderStatus.Pending,
        DetailedOrder = new DetailedOrder
        {
            Status = OrderStatus.Pending,
            ShippingAddress = "221 B Baker St, London",
            BillingAddress = "11 Wall Street, New York"
        }
    });
}

context.SaveChanges();
}

using (var context = new TableSplittingContext())
{
    var pendingCount = context.Orders.Count(o => o.Status == OrderStatus.Pending);
    Console.WriteLine($"Current number of pending orders: {pendingCount}");
}

using (var context = new TableSplittingContext())
{
    var order = context.DetailedOrders.First(o => o.Status == OrderStatus.Pending);
    Console.WriteLine($"First pending order will ship to: {order.ShippingAddress}");
}
```

可选依赖实体

NOTE

EF Core 3.0 中引入了此功能。

如果依赖实体使用的所有列都 `NULL` 在数据库中，则查询时将不会创建该实体的任何实例。这允许对一个可选的依赖实体建模，其中主体的关系属性将为 `null`。请注意，这也会导致所有依赖属性均为可选，并设置为 `null`，这可能不是预期的。

并发令牌

如果共享表的任何实体类型都有并发标记，则还必须将其包含在所有其他实体类型中。当只更新映射到同一个表中的一个实体时，必须使用此值来避免陈旧并发令牌值。

若要避免向使用代码公开并发标记，可以将其创建为 [影子属性](#)：

```
modelBuilder.Entity<Order>()
    .Property<byte[]>("Version").IsRowVersion().HasColumnName("Version");

modelBuilder.Entity<DetailedOrder>()
    .Property(o => o.Version).IsRowVersion().HasColumnName("Version");
```

从属实体类型

2020/3/11 •

EF Core 让你可以只显示对其他实体类型的导航属性的模型实体类型。它们称为_拥有的实体类型_。包含拥有的实体类型的实体是其_所有者_。

拥有的实体实质上是所有者的一部分，并且在没有它的情况下不存在，它们在概念上类似于聚合。这意味着，拥有的实体由与所有者的关系的从属方定义。

显式配置

拥有永远不会包含类型由 EF Core 模型中按照约定的实体。你可以使用中的 `OwnsOne` 方法 `OnModelCreating` 或使用 `OwnedAttribute` (EF Core 2.1 中的 new) 批注该类型，以便将该类型配置为拥有的类型。

在此示例中，`StreetAddress` 是无标识属性的类型。它用作 `Order` 类型的属性来指定特定订单的发货地址。

在从另一个实体类型引用时，可以使用 `OwnedAttribute` 将其视为拥有的实体：

```
[Owned]
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

```
public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

还可以使用 `OnModelCreating` 中的 `OwnsOne` 方法来指定 `ShippingAddress` 属性是 `Order` 实体类型的拥有实体，并根据需要配置其他方面。

```
modelBuilder.Entity<Order>().OwnsOne(p => p.ShippingAddress);
```

如果 `ShippingAddress` 属性在 `Order` 类型中是私有的，则可以使用 `OwnsOne` 方法的字符串版本：

```
modelBuilder.Entity<Order>().OwnsOne(typeof(StreetAddress), "ShippingAddress");
```

有关更多上下文，请参阅[完整的示例项目](#)。

隐式键

使用 `OwnsOne` 配置的或通过引用导航发现的拥有的类型与所有者始终具有一对一的关系，因此，它们不需要其自己的键值，因为外键值是唯一的。在上面的示例中，`StreetAddress` 类型不需要定义键属性。

为了理解 EF Core 如何跟踪这些对象，知道主键是作为所属类型的影子属性创建的，这会很有用。所拥有类型的实例的键值将与所有者实例的键的值相同。

拥有的类型的集合

NOTE

此为 EF Core 2.2 中的新增功能。

若要配置拥有的类型的集合, 请使用 `OnModelCreating` 中 `OwnsMany`。

拥有的类型需要主键。如果 .NET 类型上没有合适的候选属性, EF Core 可以尝试创建一个。但是, 当所有类型都通过集合进行定义时, 只需创建一个影子属性以同时充当所有者的外键和该拥有实例的主键, 就像在 `OwnsOne` 中一样: 对于每个所有者, 可以有多个拥有的类型实例, 因此, 所有者的密钥不足以以为每个拥有的实例提供唯一标识。

这两个最直接的解决方案是:

- 在独立于指向所有者的外键的新属性上定义代理项主键。所有所有者都需要具有唯一的值(例如, 如果父 {1} 具有子 {1}, 则父 {2} 不能具有子 {1}), 因此该值没有任何固有含义。由于外键不是主键的一部分, 因此可以更改其值, 因此, 您可以将子级从一个父级移到另一个父级, 但这通常会针对聚合语义进行。
- 使用外键和附加属性作为组合键。现在, 附加属性值只需对于给定父级是唯一的(因此, 如果父 {1} 具有子 {1,1}, 则父 {2} 仍可以具有子 {2,1})。通过创建主键的外键部分, 所有者和拥有的实体之间的关系将变为不可变的, 并且更好地反映了聚合语义。这是 EF Core 默认情况下执行的操作。

在此示例中, 我们将使用 `Distributor` 类:

```
public class Distributor
{
    public int Id { get; set; }
    public ICollection<StreetAddress> ShippingCenters { get; set; }
}
```

默认情况下, 通过 `ShippingCenters` 导航属性引用的所属类型使用的主键将 `("DistributorId", "Id")` 其中 `"DistributorId"` 为 FK, `"Id"` 为唯一的 `int` 值。

若要配置不同的 PK 调用 `HasKey`:

```
modelBuilder.Entity<Distributor>().OwnsMany(p => p.ShippingCenters, a =>
{
    a.WithOwner().HasForeignKey("OwnerId");
    a.Property<int>("Id");
    a.HasKey("Id");
});
```

NOTE

EF Core 3.0 `WithOwner()` 方法不存在, 因此应删除此调用。此外, 不会自动发现主键, 因此始终指定了它。

将拥有的类型映射到表拆分

使用关系数据库时, 默认情况下, 引用拥有的类型将映射到与所有者相同的表。这需要将表拆分为两个:某些列将用于存储所有者的数据, 某些列将用于存储拥有实体的数据。这是一种称为[表拆分](#)的常见功能。

默认情况下, EF Core 会按照模式`_Navigation_OwnedEntityProperty_`为拥有的实体类型的属性命名数据库列。因此, `StreetAddress` 属性将显示在 "Orders" 表中, 名称为 "ShippingAddress_Street" 和 "ShippingAddress_City"。

您可以使用 `HasColumnName` 方法重命名这些列:

```
modelBuilder.Entity<Order>().OwnsOne(
    o => o.ShippingAddress,
    sa =>
    {
        sa.Property(p => p.Street).HasColumnName("ShipsToStreet");
        sa.Property(p => p.City).HasColumnName("ShipsToCity");
    });
});
```

NOTE

大多数正常的实体类型配置方法(如`Ignore`)都可以通过相同的方式进行调用。

在多个所拥有的类型之间共享相同的 .NET 类型

一个拥有的实体类型可以是与另一个拥有的实体类型相同的 .NET 类型，因此，.NET 类型可能不足以标识某个所有者的类型。

在这些情况下，从所有者指向拥有的实体的属性将成为所拥有实体类型的_定义导航_。从 EF Core 的角度来看，定义导航是标识的旁边的.NET 类型的类型的一部分。

例如，在下面的类中 `ShippingAddress` 和 `BillingAddress` 均为同一 .NET 类型，`StreetAddress`：

```
public class OrderDetails
{
    public DetailedOrder Order { get; set; }
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

若要了解如何 EF Core 将区分跟踪这些对象的实例，可能会有用考虑定义导航已旁边的值的键的所有者的实例键的一部分，拥有类型的.NET 类型。

嵌套的所属类型

在此示例中 `OrderDetails` 拥有 `BillingAddress` 和 `ShippingAddress`，它们都是 `StreetAddress` 类型。然后 `OrderDetails` 归 `DetailedOrder` 类型所有。

```
public class DetailedOrder
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
    public OrderStatus Status { get; set; }
}
```

```
public enum OrderStatus
{
    Pending,
    Shipped
}
```

每个指向所拥有的类型的导航都定义一个具有完全独立配置的单独实体类型。

除了嵌套的类型外，拥有的类型还可以引用常规实体，只要拥有的实体在依赖方，就可以是所有者或其他实体。此功能在 EF6 中除复杂类型外，会设置拥有的实体类型。

```
public class OrderDetails
{
    public DetailedOrder Order { get; set; }
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

可以通过将 `OwnsOne` 方法链接到一个流畅的调用来配置此模型：

```
modelBuilder.Entity<DetailedOrder>().OwnsOne(p => p.OrderDetails, od =>
{
    od.WithOwner(d => d.Order);
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
});
```

请注意用于配置指向所有者的导航属性的 `WithOwner` 调用。若要配置不包含在所有权关系中的 owner 实体类型的导航 `WithOwner()` 应在不使用任何参数的情况下调用。

使用 `OrderDetails` 和 `StreetAddress` 上的 `OwnedAttribute` 可以获得结果。

将拥有的类型存储在单独的表中

与 EF6 复杂类型不同的是，拥有的类型可以存储在所有者的单独表中。若要重写将拥有的类型映射到与所有者相同的表的约定，只需调用 `ToTable` 并提供不同的表名即可。下面的示例将 `OrderDetails` 及其两个地址映射到不同 `DetailedOrder` 的表中：

```
modelBuilder.Entity<DetailedOrder>().OwnsOne(p => p.OrderDetails, od =>
{
    od.ToTable("OrderDetails");
});
```

还可以使用 `TableAttribute` 来实现此目的，但请注意，如果有多个导航到拥有的类型，则这会失败，因为在这种情况下，多个实体类型会映射到同一个表。

查询拥有的类型

查询所有者时，固有类型将默认包含在内。不需要使用 `Include` 方法，即使所有类型都存储在单独的表中也是如此。根据前面所述的模型，以下查询将从数据库中获取 `Order`、`OrderDetails` 和两个所拥有的 `StreetAddresses`：

```
var order = context.DetailedOrders.First(o => o.Status == OrderStatus.Pending);
Console.WriteLine($"First pending order will ship to: {order.OrderDetails.ShippingAddress.City}");
```

限制

其中一些限制对于拥有的实体类型的工作方式很重要，但其他一些限制是我们可以在未来版本中删除的限制：

按设计限制

- 不能为拥有的类型创建 `DbSet<T>`
- 无法在 `ModelBuilder` 上使用拥有的类型调用 `Entity<T>()`

当前缺陷

- 拥有的实体类型不能具有继承层次结构
- 引用导航到拥有的实体类型不能为 null，除非它们显式映射到与所有者不同的表

- 拥有的实体类型的实例不能由多个所有者共享(这是一个已知的值对象方案, 不能使用拥有的实体类型来实现)

以前版本中的缺点

- 在 EF Core 2.0 中, 在派生的实体类型中不能声明导航到拥有的实体类型, 除非将拥有的实体显式映射到所有者层次结构中的单独表。此限制已在 EF Core 2.1 中被删除
- 在 EF Core 仅支持2.0 和2.1 的引用导航到拥有的类型。此限制已在 EF Core 2.2 中被删除

无键实体类型

2020/3/11 •

NOTE

此功能已添加到 EF Core 2.1 的查询类型的名称下。在 EF Core 3.0 中，概念已重命名为无键实体类型。

除了常规实体类型外，EF Core 模型还可以包含_无键实体类型_，可用于对不包含键值的数据执行数据库查询。

无键实体类型特征

无键实体类型支持与常规实体类型相同的多个映射功能，如继承映射和导航属性。上关系存储，他们可以配置的目标数据库对象和列通过 fluent API 方法或数据注释。

但是，它们不同于常规实体类型，因为它们：

- 不能定义键。
- 永远不会对`_DbContext_`中的更改进行跟踪，因此不会在数据库中插入、更新或删除这些更改。
- 永远不会由约定发现。
- 仅支持导航映射功能的子集，具体如下：
 - 它们可能永远不会作为关系的主体端。
 - 它们可能没有到拥有的实体的导航
 - 它们只能包含指向常规实体的引用导航属性。
 - 实体不能包含无键实体类型的导航属性。
- 需要配置`.HasNoKey()`方法调用。
- 可以映射到定义的_查询_。定义查询是在模型中声明的查询，它充当无键实体类型的数据源。

使用方案

无键实体类型的一些主要使用方案包括：

- 作为[原始 SQL 查询](#)的返回类型。
- 映射到不包含主键的数据库视图。
- 映射到不具有定义的主键的表。
- 映射到模型中定义的查询。

映射到数据库对象

将无键实体类型映射到数据库对象是使用`ToTable`或`ToView`Fluent API 实现的。从 EF Core 的角度来看，此方法中指定的数据库对象是一个_视图_，这意味着它将被视为只读查询源，并且不能作为更新、插入或删除操作的目标。但是，这并不意味着数据库对象实际上必须是数据库视图。它也可以是将被视为只读的数据库表。相反，对于常规实体类型，EF Core 假设在`ToTable`方法中指定的数据库对象可以视为_表_，这意味着它可用作查询源，但也可作为更新、删除和插入操作的目标。事实上，您可以在`ToTable`中指定数据库视图的名称，只要该视图配置为可在数据库上更新，一切都应正常运行。

NOTE

`ToView`假设对象已存在于数据库中，并且不是由迁移创建的。

示例

下面的示例演示如何使用无键实体类型来查询数据库视图。

TIP

可在 GitHub 上查看此文章的[示例](#)。

首先，我们定义一个简单的博客和文章模型：

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
}
```

接下来，我们定义一个简单的数据库视图，这样就可以查询与每个博客帖子数：

```
db.Database.ExecuteSqlRaw(
    @"CREATE VIEW View_BlogPostCounts AS
        SELECT b.Name, Count(p.PostId) as PostCount
        FROM Blogs b
        JOIN Posts p on p.BlogId = b.BlogId
        GROUP BY b.Name");
```

接下来，我们定义一个类来保存数据库视图的结果：

```
public class BlogPostsCount
{
    public string BlogName { get; set; }
    public int PostCount { get; set; }
}
```

接下来，使用 `HasNoKey` API 在 `_OnModelCreating_` 中配置无键实体类型。我们使用熟知的配置 API 来配置无键实体类型的映射：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<BlogPostsCount>(eb =>
    {
        eb.HasKey();
        eb.ToView("View_BlogPostCounts");
        eb.Property(v => v.BlogName).HasColumnName("Name");
    });
}
```

接下来，将 `DbContext` 配置为包括 `DbSet<T>`：

```
public DbSet<BlogPostsCount> BlogPostCounts { get; set; }
```

最后，我们可以采用标准方式来查询数据库视图：

```
var postCounts = db.BlogPostCounts.ToList();

foreach (var postCount in postCounts)
{
    Console.WriteLine($"{postCount.BlogName} has {postCount.PostCount} posts.");
    Console.WriteLine();
}
```

TIP

请注意，我们还定义了一个上下文级查询属性(DbSet)作为此类型的查询的根。

在具有相同 DbContext 类型的多个模型之间交替

2020/3/11 •

OnModelCreating 中生成的模型可以使用上下文中的属性来更改模型的生成方式。例如，假设你想要根据某些属性以不同的方式配置实体：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    if (UseIntProperty)
    {
        modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.StringProperty);
    }
    else
    {
        modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.IntProperty);
    }
}
```

遗憾的是，此代码不会按原样工作，因为 EF 会构建模型并只运行一次 OnModelCreating，出于性能原因缓存结果。但是，可以挂钩到模型缓存机制，使 EF 知道生成不同模型的属性。

IModelCacheKeyFactory

EF 使用 IModelCacheKeyFactory 来生成模型的缓存键；默认情况下，EF 假设对于任何给定的上下文类型，模型都是相同的，因此该服务的默认实现将返回只包含上下文类型的键。若要从相同的上下文类型生成不同的模型，需要将 IModelCacheKeyFactory 服务替换为正确的实现；将使用 Equals 方法将生成的键与其他模型键进行比较，并考虑影响模型的所有变量：

生成模型缓存密钥时，以下实现会考虑 IgnoreIntProperty：

```
public class DynamicModelCacheKeyFactory : IModelCacheKeyFactory
{
    public object Create(DbContext context)
        => context is DynamicContext dynamicContext
            ? (context.GetType(), dynamicContext.UseIntProperty)
            : (object)context.GetType();
}
```

最后，在上下文的 OnConfiguring 中注册新 IModelCacheKeyFactory：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseInMemoryDatabase("DynamicContext")
        .ReplaceService<IModelCacheKeyFactory, DynamicModelCacheKeyFactory>();
```

有关更多上下文，请参阅[完整的示例项目](#)。

空间数据

2020/3/11 ·

NOTE

此功能是在 EF Core 2.2 中添加的。

空间数据表示对象的物理位置和形状。许多数据库提供对此类数据的支持，以便能够与其他数据一起进行索引和查询。常见方案包括从位置在给定距离内查询对象，或选择其边框包含给定位置的对象。EF Core 支持使用[NetTopologySuite](#)空间库映射到空间数据类型。

安装

若要在 EF Core 中使用空间数据，需要安装相应的支持 NuGet 包。需要安装哪个包取决于所使用的提供程序。

EF CORE	NUGET
Microsoft.EntityFrameworkCore.SqlServer	Microsoft.entityframeworkcore. NetTopologySuite
Microsoft.EntityFrameworkCore.Sqlite	Microsoft.entityframeworkcore. NetTopologySuite
Microsoft.EntityFrameworkCore.InMemory	NetTopologySuite
Npgsql.EntityFrameworkCore.PostgreSQL	Npgsql. Microsoft.entityframeworkcore. PostgreSQL. NetTopologySuite

反向工程

空间 NuGet 包还启用具有空间属性的反向工程模型，但需要在运行 `Scaffold-DbContext` 或 `dotnet ef dbcontext scaffold` 之前安装包。否则，你将收到有关找不到列的类型映射的警告，将跳过这些列。

NetTopologySuite (NTS)

NetTopologySuite 是用于 .NET 的空间库。EF Core 使用模型中的 NTS 类型启用映射到数据库中的空间数据类型。

若要通过 NTS 启用到空间类型的映射，请在提供程序的 DbContext 选项生成器上调用 `UseNetTopologySuite` 方法。例如，对于 SQL Server，你应将其称为。

```
optionsBuilder.UseSqlServer(
    @"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=WideWorldImporters",
    x => x.UseNetTopologySuite());
```

有几种空间数据类型。使用哪种类型取决于您想要允许的形状的类型。下面是可用于模型中的属性的 NTS 类型的层次结构。它们位于 `NetTopologySuite.Geometries` 命名空间中。

- Geometry
 - 点
 - LineString
 - Polygon

- GeometryCollection
 - MultiPoint
 - MultiLineString
 - MultiPolygon

WARNING

NTS 不支持 CircularString、CompoundCurve 和 CurvePolygon。

使用基本几何图形类型允许属性指定任意类型的形状。

以下实体类可用于映射到[广角导入示例数据库](#)中的表。

```
[Table("Cities", Schema = "Application")]
class City
{
    public int CityID { get; set; }

    public string CityName { get; set; }

    public Point Location { get; set; }
}

[Table("Countries", Schema = "Application")]
class Country
{
    public int CountryID { get; set; }

    public string CountryName { get; set; }

    // Database includes both Polygon and MultiPolygon values
    public Geometry Border { get; set; }
}
```

创建值

您可以使用构造函数来创建 geometry 对象;但是, NTS 建议改为使用几何工厂。这允许您指定默认 SRID (坐标使用的空间引用系统), 并使您能够控制更高级的任务(例如, 在计算过程中使用)和坐标序列(确定哪些坐标维度和度量值--可用)。

```
var geometryFactory = NtsGeometryServices.Instance.CreateGeometryFactory(srid: 4326);
var currentLocation = geometryFactory.CreatePoint(-122.121512, 47.6739882);
```

NOTE

4326指的是 WGS 84, 是 GPS 和其他地理系统中使用的标准。

经度和纬度

NTS 中的坐标采用 X 和 Y 值。若要表示经度和纬度, 请将 X 用于经度, 将 Y 用于纬度。请注意, 这是从通常会看到这些值的 `latitude, longitude` 格式反向进行的。

在客户端操作过程中忽略 SRID

NTS 在操作过程中忽略 SRID 值。它假定为平面坐标系统。这意味着, 如果在经度和纬度方面指定了坐标, 则某些客户端计算的值(例如, 距离、长度和区域)将为度数, 而不是计量。若要获得更有意义的值, 首先需要使用库(如[ProjNet4GeoAPI](#))在计算这些值之前投影到另一个坐标系统的坐标。

如果通过 EF Core 通过 SQL 对操作进行服务器计算，则该结果的单元将由数据库确定。

下面是一个示例，说明如何使用 ProjNet4GeoAPI 来计算两个城市之间的距离。

```
static class GeometryExtensions
{
    static readonly CoordinateSystemServices _coordinateSystemServices
        = new CoordinateSystemServices(
            new CoordinateSystemFactory(),
            new CoordinateTransformationFactory(),
            new Dictionary<int, string>
            {
                // Coordinate systems:

                [4326] = GeographicCoordinateSystem.WGS84.WKT,
                // This coordinate system covers the area of our data.
                // Different data requires a different coordinate system.
                [2855] =
                @"
                    PROJCS[""NAD83(HARN) / Washington North"",
                        GEOGCS[""NAD83(HARN)"",
                            DATUM[""NAD83_High_Accuracy_Regional_Network"",
                                SPHEROID[""GRS 1980"",6378137,298.257222101,
                                    AUTHORITY[""EPSG"",""7019""}],
                                AUTHORITY[""EPSG"",""6152""}],
                            PRIMEM[""Greenwich"",0,
                                AUTHORITY[""EPSG"",""8901""]],
                            UNIT[""degree"",0.01745329251994328,
                                AUTHORITY[""EPSG"",""9122""]],
                                AUTHORITY[""EPSG"",""4152""],
                                PROJECTION[""Lambert_Conformal_Conic_2SP""],
                                PARAMETER[""standard_parallel_1"",48.73333333333333],
                                PARAMETER[""standard_parallel_2"",47.5],
                                PARAMETER[""latitude_of_origin"",47],
                                PARAMETER[""central_meridian"", -120.833333333333],
                                PARAMETER[""false_easting"",500000],
                                PARAMETER[""false_northing"",0],
                                UNIT[""metre"",1,
                                    AUTHORITY[""EPSG"",""9001""]],
                                    AUTHORITY[""EPSG"",""2855""]
                                ]
                            );
                }
            });

    public static Geometry ProjectTo(this Geometry geometry, int srid)
    {
        var transformation = _coordinateSystemServices.CreateTransformation(geometry.SRID, srid);

        var result = geometry.Copy();
        result.Apply(new MathTransformFilter(transformation.MathTransform));

        return result;
    }

    class MathTransformFilter : ICoordinateSequenceFilter
    {
        readonly MathTransform _transform;

        public MathTransformFilter(MathTransform transform)
            => _transform = transform;

        public bool Done => false;
        public bool GeometryChanged => true;

        public void Filter(CoordinateSequence seq, int i)
        {
            var result = _transform.Transform(
                new[] { seq[i] });
        }
    }
}
```

```
        {
            seq.GetOrdinate(i, Ordinate.X),
            seq.GetOrdinate(i, Ordinate.Y)
        });
        seq.SetOrdinate(i, Ordinate.X, result[0]);
        seq.SetOrdinate(i, Ordinate.Y, result[1]);
    }
}
}
```

```
var seattle = new Point(-122.333056, 47.609722) { SRID = 4326 };
var redmond = new Point(-122.123889, 47.669444) { SRID = 4326 };

var distance = seattle.ProjectTo(2855).Distance(redmond.ProjectTo(2855));
```

查询数据

在 LINQ 中，可用作数据库函数的 NTS 方法和属性将转换为 SQL。例如，在以下查询中转换距离和包含方法。本文末尾的表格显示了不同 EF Core 提供商支持哪些成员。

```
var nearestCity = db.Cities
    .OrderBy(c => c.Location.Distance(currentLocation))
    .FirstOrDefault();

var currentCountry = db.Countries
    .FirstOrDefault(c => c.Border.Contains(currentLocation));
```

SQL Server

如果你正在使用 SQL Server，你还应该注意一些其他问题。

地理或几何图形

默认情况下，空间属性映射到 SQL Server 中的 `geography` 列。若要使用 `geometry`，请在模型中[配置列类型](#)。

地理多边形环

使用 `geography` 列类型时，SQL Server 对外环（或外壳）和内部环（或孔）施加附加要求。外部环必须逆时针旋转，并顺时针旋转内部环。NTS 在将值发送到数据库之前对其进行验证。

FullGlobe

使用 `geography` 列类型时，SQL Server 具有非标准几何类型来表示全地球。它还提供了一种方法，用于根据全地球（无外部环）来表示多边形。NTS 不支持这两种方法。

WARNING

NTS 不支持基于 FullGlobe 和多边形。

SQLite

下面是使用 SQLite 的一些其他信息。

安装 SpatiaLite

在 Windows 上，本机 `mod_spatialite` 库以 NuGet 包的依赖项的形式分发。其他平台需要单独安装它。通常使用软件包管理器完成此操作。例如，可以在 Ubuntu 上使用 APT 和 MacOS 上的 Homebrew。

```
# Ubuntu
apt-get install libsqlite3-mod-spatialite

# macOS
brew install libspatialite
```

遗憾的是，较新版本的 PROJ (SpatiaLite 的依赖项) 与 EF 的默认 [SQLitePCLRaw 绑定](#) 不兼容。若要解决此情况，可以创建使用系统 SQLite 库的自定义 [SQLitePCLRaw 提供程序](#)，也可以安装 SPATIALITE 禁用 PROJ 支持的自定义生成。

```
curl https://www.gaia-gis.it/gaia-sins/libspatialite-4.3.0a.tar.gz | tar -xz
cd libspatialite-4.3.0a

if [[ `uname -s` == Darwin* ]]; then
    # Mac OS requires some minor patching
    sed -i "" "s/shrext_cmds='`test \\.\\$module = .yes && echo .so \\|\\`' echo \\.dylib`'/shrext_cmds='\\.dylib'/" configure
fi

./configure --disable-proj
make
make install
```

配置 SRID

在 SpatiaLite 中，列需要为每个列指定一个 SRID。默认的 SRID 是 `0`。使用 `ForSqliteHasSrid` 方法指定其他 SRID。

```
modelBuilder.Entity<City>().Property(c => c.Location)
    .ForSqliteHasSrid(4326);
```

Dimension

类似于 SRID，列的维度（或坐标）也被指定为列的一部分。默认坐标为 X 和 Y。使用 `ForSqliteHasDimension` 方法启用其他坐标（Z 和 M）。

```
modelBuilder.Entity<City>().Property(c => c.Location)
    .ForSqliteHasDimension(Ordinates.XYZ);
```

转换的操作

此表显示每个 EF Core 提供程序将哪些 NTS 成员转换为 SQL。

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQlite	Npgsql
Geometry	✓	✓	✓	✓
AsBinary ()	✓	✓	✓	✓
AsText ()	✓	✓	✓	✓
Geometry	✓		✓	✓
Geometry (双精度型)	✓	✓	✓	✓

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQlite	Npgsql
Geometry (double, int)			✓	✓
质心	✓		✓	✓
Geometry。Contains (Geometry)	✓	✓	✓	✓
ConvexHull ()	✓	✓	✓	✓
CoveredBy (Geometry)			✓	✓
Geometry (Geometry)			✓	✓
几何。交叉(几何)	✓		✓	✓
几何差(几何)	✓	✓	✓	✓
Geometry。维度	✓	✓	✓	✓
不连续(Geometry)	✓	✓	✓	✓
Geometry (Geometry)	✓	✓	✓	✓
Geometry 信封	✓		✓	✓
EqualsExact (Geometry)				✓
EqualsTopologically (Geometry)	✓	✓	✓	✓
GeometryType	✓	✓	✓	✓
GetGeometryN (int)	✓		✓	✓
InteriorPoint	✓		✓	✓
几何交集(Geometry)	✓	✓	✓	✓
几何和交集 (Geometry)	✓	✓	✓	✓
IsEmpty	✓	✓	✓	✓
IsSimple	✓		✓	✓
Geometry	✓	✓	✓	✓

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQlite	Npgsql
IsWithinDistance (Geometry, double)	✓		✓	✓
Geometry。长度	✓	✓	✓	✓
NumGeometries	✓	✓	✓	✓
X.numpoints	✓	✓	✓	✓
OgcGeometryType	✓	✓	✓	✓
Geometry 重叠 (Geometry)	✓	✓	✓	✓
PointOnSurface	✓		✓	✓
Geometry (Geometry, string)	✓		✓	✓
Geometry 反向()			✓	✓
SRID	✓	✓	✓	✓
SymmetricDifference (Geometry)	✓	✓	✓	✓
ToBinary ()	✓	✓	✓	✓
ToText ()	✓	✓	✓	✓
几何图形(几何)	✓		✓	✓
Geometry ()			✓	✓
Geometry (Geometry)	✓	✓	✓	✓
几何图形。内 (Geometry)	✓	✓	✓	✓
GeometryCollection	✓	✓	✓	✓
GeometryCollection [int]	✓	✓	✓	✓
LineString	✓	✓	✓	✓
LineString 终结点	✓	✓	✓	✓
LineString. GetPointN (int)	✓	✓	✓	✓

NETTOPOLOGYSUITE	SQL SERVER (GEOMETRY)	SQL SERVER (GEOGRAPHY)	SQlite	Npgsql
LineString. IsClosed	✓	✓	✓	✓
LineString. IsRing	✓		✓	✓
LineString. StartPoint	✓	✓	✓	✓
MultiLineString. IsClosed	✓	✓	✓	✓
点 M	✓	✓	✓	✓
点 X	✓	✓	✓	✓
Point。Y	✓	✓	✓	✓
点 Z	✓	✓	✓	✓
多边形。ExteriorRing	✓	✓	✓	✓
GetInteriorRingN (int)	✓	✓	✓	✓
多边形。 NumInteriorRings	✓	✓	✓	✓

其他资源

- [SQL Server 中的空间数据](#)
- [SpatiaLite 主页](#)
- [Npgsql 空间文档](#)
- [PostGIS 文档](#)

管理数据库架构

2020/4/8 •

EF Core 提供两种主要方法来保持 EF Core 模型和数据库架构同步。至于我们应该选用哪个方法, 请确定你是希望以 EF Core 模型为准还是以数据库为准。

如果希望以 EF Core 模型为准, 请使用[迁移](#)。对 EF Core 模型进行更改时, 此方法会以增量方式将相应架构更改应用到数据库, 以使数据库保持与 EF Core 模型兼容。

如果希望以数据库架构为准, 请使用[反向工程](#)。使用此方法, 可通过将数据库架构反向工程到 EF Core 模型来生成相应的 DbContext 和实体类型。

NOTE

[创建和删除 API](#) 也可从 EF Core 模型创建数据库架构。但是, 它们主要适用于允许删除数据库的场景, 比如测试、原型制作等。

迁移

2020/4/8 •

开发期间，数据模型将发生更改并与数据库不同步。可以删除该数据库，让 EF 创建一个新的数据库来匹配该模型，但此过程会导致数据丢失。EF Core 中的迁移功能能够以递增方式更新数据库架构，使其与应用程序的数据模型保持同步，同时保留数据库中的现有数据。

迁移包括命令行工具和 API，可帮助执行以下任务：

- [创建迁移](#)。生成可以更新数据库以使其与一系列模型更改同步的代码。
- [更新数据库](#)。应用挂起的迁移更新数据库架构。
- [自定义迁移代码](#)。有时，需要修改或补充生成的代码。
- [删除迁移](#)。删除生成的代码。
- [还原迁移](#)。撤消数据库更改。
- [生成 SQL 脚本](#)。可能需要一个脚本来更新生产数据库，或者对迁移代码进行故障排除。
- [在运行时应用迁移](#)。当设计时更新和正在运行脚本不是最佳选项时，调用 `Migrate()` 方法。

TIP

如果 `DbContext` 与启动项目位于不同程序集中，可以在[包管理器控制台工具](#)或[.NET Core CLI 工具](#)中显式指定目标和启动项目。

安装工具

安装[命令提示符工具](#)：

- 对于 Visual Studio，建议使用[包管理器控制台工具](#)。
- 对于其他开发环境，请选择[.NET Core CLI 工具](#)。

创建迁移

定义[初始模型](#)后，即应创建数据库。若要添加初始迁移，请运行以下命令。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef migrations add InitialCreate
```

向[Migrations](#)目录下的项目添加以下三个文件：

- XXXXXXXXXXXXXXXX_InitialCreate.cs - 主迁移文件。包含应用迁移所需的操作（在 `Up()` 中）和还原迁移所需的操作（在 `Down()` 中）。
- XXXXXXXXXXXXXXXX_InitialCreate.Designer.cs - 迁移元数据文件。包含 EF 所用的信息。
- MyContextModelSnapshot.cs -- 当前模型的快照。用于确定添加下一迁移时的更改内容。

文件名中的时间戳有助于保持文件按时间顺序排列，以便你可以查看更改进展。

TIP

可以自由移动“Migrations”目录下的迁移文件并更改其命名空间。新建的迁移和上个迁移同级。

更新数据库

接下来，将迁移应用到数据库以创建架构。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef database update
```

自定义迁移代码

更改 EF Core 模型后，数据库架构可能不同步。为使其保持最新，请再添加一个迁移。迁移名称的用途与版本控制系统中的提交消息类似。例如，如果更改对于评审是一个新的实体类，可以选择一个名称，如 AddProductReviews。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef migrations add AddProductReviews
```

搭建迁移基架(为其生成代码)后，检查代码的准确性，并添加、删除或修改正确应用代码所需的任何操作。

例如，迁移可能包含以下操作：

```
migrationBuilder.DropColumn(  
    name: "FirstName",  
    table: "Customer");  
  
migrationBuilder.DropColumn(  
    name: "LastName",  
    table: "Customer");  
  
migrationBuilder.AddColumn<string>(  
    name: "Name",  
    table: "Customer",  
    nullable: true);
```

虽然这些操作可使数据库架构兼容，但是它们不会保留现有客户姓名。如下所示，我们可以重写以改善这一情况。

```
migrationBuilder.AddColumn<string>(
    name: "Name",
    table: "Customer",
    nullable: true);

migrationBuilder.Sql(
@"
    UPDATE Customer
    SET Name = FirstName + ' ' + LastName;
");

migrationBuilder.DropColumn(
    name: "FirstName",
    table: "Customer");

migrationBuilder.DropColumn(
    name: "LastName",
    table: "Customer");
```

TIP

当某个操作可能会导致数据丢失(例如删除某列), 搭建迁移基架过程将对此发出警告。如果看到此警告, 务必检查迁移代码的准确性。

使用相应命令将迁移应用到数据库。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef database update
```

空迁移

有时模型未变更, 直接添加迁移也很有用处。在这种情况下, 添加新迁移会创建一个带空类的代码文件。可以自定义此迁移, 执行与 EF Core 模型不直接相关的操作。可能需要通过此方式管理的一些事项包括:

- 全文搜索
- 函数
- 存储过程
- 触发器
- 视图

删除迁移

有时, 你可能在添加迁移后意识到需要在应用迁移前对 EF Core 模型作出其他更改。要删除上个迁移, 请使用如下命令。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef migrations remove
```

删除迁移后, 可对模型作出其他更改, 然后再次添加迁移。

还原迁移

如果已对数据库应用一个迁移(或多个迁移),但需将其复原,则可使用同一命令来应用迁移,并指定回退时的目标迁移名称。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef database update LastGoodMigration
```

生成 SQL 脚本

调试迁移或将其部署到生产数据库时,生成一个 SQL 脚本很有帮助。之后可进一步检查该脚本的准确性,并对其作出调整以满足生产数据库的需求。该脚本还可与部署技术结合使用。基本命令如下。

- [.NET Core CLI](#)
- [Visual Studio](#)

基本用法

```
dotnet ef migrations script
```

使用 `From`(`to` 隐含)

这将生成从此迁移到最新迁移的 SQL 脚本。

```
dotnet ef migrations script 20190725054716_Add_new_tables
```

使用 `From` 和 `To`

这将生成从 `from` 迁移到指定 `to` 迁移的 SQL 脚本。

```
dotnet ef migrations script 20190725054716_Add_new_tables 20190829031257_Add_audit_table
```

可以使用比 `to` 新的 `from` 来生成回退脚本。请记下潜在的数据丢失方案。

此命令有几个选项。

`from` 迁移应是运行该脚本前应用到数据库的最后一个迁移。如果未应用任何迁移,请指定 `0`(默认值)。

`to` 迁移是运行该脚本后应用到数据库的最后一个迁移。它默认为项目中的最后一个迁移。

可以选择生成 `idempotent` 脚本。此脚本仅会应用尚未应用到数据库的迁移。如果不确知应用到数据库的最后一个迁移或需要部署到多个可能分别处于不同迁移的数据库,此脚本非常有用。

在运行时应用迁移

启动或首次运行期间,一些应用可能需要在运行时应用迁移。为此,请使用 `Migrate()` 方法。

此方法构建于 `IMigrator` 服务之上,该服务可用于更多高级方案。请使用
`myDbContext.GetInfrastructure().GetService<IMigrator>()` 进行访问。

```
myDbContext.Database.Migrate();
```

WARNING

- 此方法并不适合所有人。尽管此方法非常适合具有本地数据库的应用，但是大多数应用程序需要更可靠的部署策略，例如生成 SQL 脚本。
- 请勿在 `Migrate()` 前调用 `EnsureCreated()`。`EnsureCreated()` 会绕过迁移创建架构，这会导致 `Migrate()` 失败。

后续步骤

有关详细信息，请参阅 [Entity Framework Core 工具参考 - EF Core](#)。

团队环境中的迁移

2020/3/11 •

在团队环境中使用迁移时，需额外注意模型快照文件。该文件会告诉你团队成员的迁移是否能够与你的迁移顺利合并，以及你是否需要通过重新创建迁移来解决代码冲突，然后再进行共享。

合并

合并团队成员的迁移时，可能会在模型快照文件中出现冲突。如果两边的变更不相关联且合并规模较小，则两个迁移可以共存。例如，可能会在客户实体类型配置中出现如下所示的合并冲突：

```
<<<<< Mine
b.Property<bool>("Deactivated");
=====
b.Property<int>("LoyaltyPoints");
>>>>> Theirs
```

由于这两个属性需要在最终模型中共存，因此请同时添加这两个属性来完成合并。在许多情况下，版本控制系统可能会自动合并此类更改。

```
b.Property<bool>("Deactivated");
b.Property<int>("LoyaltyPoints");
```

在此类情况下，你的迁移和团队成员的迁移是相互独立的。由于可以首先应用任一迁移，因此在与团队共享你的迁移之前，无需对其进行更多的更改。

解决冲突

有时会在合并模型快照文件的过程中遇到真正的冲突。例如，你和团队成员可能重命名了同一属性。

```
<<<<< Mine
b.Property<string>("Username");
=====
b.Property<string>("Alias");
>>>>> Theirs
```

如果遇到这种冲突，请通过重新创建你的迁移来解决。执行以下步骤：

1. 中止合并，回退到合并前的工作目录
2. 删除你的迁移（但保留你的模型更改）
3. 将团队成员的更改合并到你的工作目录
4. 重新添加迁移

执行此操作后，可以按正确的顺序应用两个迁移。首先应用其迁移，将列重命名为*Alias*，此后，迁移会将其重命名为*Username*。

你的迁移可以安全地与团队的其他人共享。

自定义迁移操作

2020/3/11 ·

借助 MigrationBuilder API，你可以在迁移过程中执行多种不同的操作，但远远超出了这一过程。不过，API 也可通过扩展来定义自己的操作。可以通过两种方法来扩展 API：使用 `Sql()` 方法，或者通过定义自定义 `MigrationOperation` 对象。

为了说明这一点，让我们看一下如何实现使用每种方法创建数据库用户的操作。在我们的迁移中，我们希望能够编写以下代码：

```
migrationBuilder.CreateUser("SQLUser1", "Password");
```

使用 MigrationBuilder ()

实现自定义操作的最简单方法是定义调用 `MigrationBuilder.Sql()` 的扩展方法。下面是生成相应 Transact-sql 的示例。

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
=> migrationBuilder.Sql($"CREATE USER {name} WITH PASSWORD '{password}';");
```

如果迁移需要支持多个数据库提供程序，则可以使用 "`MigrationBuilder.ActiveProvider`" 属性。下面是同时支持 Microsoft SQL Server 和 PostgreSQL 的示例。

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
{
    switch (migrationBuilder.ActiveProvider)
    {
        case "Npgsql.EntityFrameworkCore.PostgreSQL":
            return migrationBuilder
                .Sql($"CREATE USER {name} WITH PASSWORD '{password}';");

        case "Microsoft.EntityFrameworkCore.SqlServer":
            return migrationBuilder
                .Sql($"CREATE USER {name} WITH PASSWORD = '{password}';");
    }

    return migrationBuilder;
}
```

此方法仅在知道将应用自定义操作的每个提供程序时才有效。

使用 MigrationOperation

若要将自定义操作与 SQL 分离，可以定义自己的 `MigrationOperation` 来表示它。然后，将操作传递给提供程序，以便它可以确定要生成的相应 SQL。

```
class CreateUserOperation : MigrationOperation
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

利用此方法，扩展方法只需将其中一个操作添加到 `MigrationBuilder.Operations`。

```
static MigrationBuilder CreateUser(
    this MigrationBuilder migrationBuilder,
    string name,
    string password)
{
    migrationBuilder.Operations.Add(
        new CreateUserOperation
        {
            Name = name,
            Password = password
        });

    return migrationBuilder;
}
```

此方法要求每个提供程序都知道如何在 `IMigrationsSqlGenerator` 服务中为此操作生成 SQL。下面是一个示例，用于重写 SQL Server 的生成器来处理新操作。

```

class MyMigrationsSqlGenerator : SqlServerMigrationsSqlGenerator
{
    public MyMigrationsSqlGenerator(
        MigrationsSqlGeneratorDependencies dependencies,
        IMigrationsAnnotationProvider migrationsAnnotations)
        : base(dependencies, migrationsAnnotations)
    {
    }

    protected override void Generate(
        MigrationOperation operation,
        IModel model,
        MigrationCommandListBuilder builder)
    {
        if (operation is CreateUserOperation createUserOperation)
        {
            Generate(createUserOperation, builder);
        }
        else
        {
            base.Generate(operation, model, builder);
        }
    }

    private void Generate(
        CreateUserOperation operation,
        MigrationCommandListBuilder builder)
    {
        var sqlHelper = Dependencies.SqlGenerationHelper;
        var stringMapping = Dependencies.TypeMappingSource.FindMapping(typeof(string));

        builder
            .Append("CREATE USER ")
            .Append(sqlHelper.DelimitIdentifier(operation.Name))
            .Append(" WITH PASSWORD = ")
            .Append(stringMapping.GenerateSqlLiteral(operation.Password))
            .AppendLine(sqlHelper.StatementTerminator)
            .EndCommand();
    }
}

```

将默认迁移 sql 生成器服务替换为已更新的。

```

protected override void OnConfiguring(DbContextOptionsBuilder options)
=> options
    .UseSqlServer(connectionString)
    .ReplaceService<IMigrationsSqlGenerator, MyMigrationsSqlGenerator>();

```

使用单独的迁移项目

2020/3/11 •

你可能想要将迁移存储在与包含你的 `DbContext` 的程序集不同的程序集中。你还可以使用此策略来维护多个迁移集，例如，一个用于开发，另一个用于发布到发布升级。

为此，请执行以下操作...

1. 创建一个新的类库。
2. 添加对 `DbContext` 程序集的引用。
3. 将迁移和模型快照文件移动到类库。

TIP

如果没有现有迁移，请在包含 `DbContext` 的项目中生成一个迁移，然后移动它。这一点很重要，因为如果迁移程序集不包含现有迁移，则添加迁移命令将无法找到 `DbContext`。

4. 配置迁移程序集：

```
options.UseSqlServer(  
    connectionString,  
    x => x.MigrationsAssembly("MyApp.Migrations"));
```

5. 从启动程序集添加对迁移程序集的引用。

- 如果这导致循环依赖项，请更新类库的输出路径：

```
<PropertyGroup>  
  <OutputPath>..\MyStartupProject\bin\$(Configuration)\</OutputPath>  
</PropertyGroup>
```

如果一切正常，应能够向项目添加新的迁移。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef migrations add NewMigration --project MyApp.Migrations
```

迁移多个提供程序

2020/3/11 •

EF Core 工具仅基架活动提供程序的迁移。但有时，您可能想要将多个提供程序（例如 Microsoft SQL Server 和 SQLite）用于 DbContext。可以通过两种方式来处理迁移。您可以维护两组迁移-每个提供程序一个，或将它们合并到可以同时使用的单个集。

两个迁移集

第一种方法是为每个模型更改生成两个迁移。

实现此目的的一种方法是将每个迁移集放在[单独的程序集中](#)，并在添加两个迁移之间手动切换活动提供程序（和迁移程序集）。

更轻松地使用工具的另一种方法是创建一个从 DbContext 派生的新类型并重写活动提供程序。此类型在设计时用于添加或应用迁移。

```
class MysqliteDbContext : MyDbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=my.db");
}
```

NOTE

由于每个迁移集使用自己的 DbContext 类型，因此此方法不需要使用单独的迁移程序集。

添加新迁移时，指定上下文类型。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet ef migrations add InitialCreate --context MyDbContext --output-dir Migrations/SqlServerMigrations
dotnet ef migrations add InitialCreate --context MysqliteDbContext --output-dir Migrations/SqliteMigrations
```

TIP

无需指定输出目录即可进行后续迁移，因为它们是以同级方式创建的。

一个迁移集

如果不喜欢两组迁移，可以手动将它们合并为可应用于这两个提供程序的单个集。

批注可以共存，因为提供程序会忽略它不理解的任何批注。例如，使用 Microsoft SQL Server 和 SQLite 的主键列可能如下所示。

```
Id = table.Column<int>(nullable: false)
    .Annotation("SqlServer:ValueGenerationStrategy",
       SqlServerValueGenerationStrategy.IdentityColumn)
    .Annotation("Sqlite:Autoincrement", true),
```

如果只能在一个提供程序上应用操作(或在提供程序之间以不同方式进行操作),请使用 `ActiveProvider` 属性告诉哪个提供程序处于活动状态。

```
if (migrationBuilder.ActiveProvider == "Microsoft.EntityFrameworkCore.SqlServer")
{
    migrationBuilder.CreateSequence(
        name: "EntityFrameworkHiLoSequence");
}
```

自定义迁移历史记录表

2020/3/11 •

默认情况下，EF Core 通过在名为 `__EFMigrationsHistory` 的表中记录哪些迁移已应用到数据库中。由于各种原因，你可能需要自定义此表以更好地满足你的需求。

IMPORTANT

如果在应用迁移后自定义迁移历史记录表，则需要负责更新数据库中的现有表。

架构和表名称

你可以使用 `OnConfiguring()` 中的 `MigrationsHistoryTable()` 方法（或 ASP.NET Core 上的 `ConfigureServices()`）更改架构和表名。下面是一个示例，说明如何使用 SQL Server EF Core 提供程序。

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlServer(
        connectionString,
        x => x.MigrationsHistoryTable("__MyMigrationsHistory", "mySchema"));
```

其他更改

若要配置表的其他方面，请重写并替换特定于提供程序的 `IHistoryRepository` 服务。下面是将 `MigrationId` 列名称更改为 SQL Server 上的 `Id` 的示例。

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IHistoryRepository, MyHistoryRepository>();
```

WARNING

`SqlServerHistoryRepository` 位于内部命名空间内部，并且在将来的版本中可能会更改。

```
class MyHistoryRepository : SqlServerHistoryRepository
{
    public MyHistoryRepository(HistoryRepositoryDependencies dependencies)
        : base(dependencies)
    {
    }

    protected override void ConfigureTable(EntityTypeBuilder<HistoryRow> history)
    {
        base.ConfigureTable(history);

        history.Property(h => h.MigrationId).HasColumnName("Id");
    }
}
```

创建和删除 API

2020/3/11 •

EnsureCreated 和 EnsureDeleted 方法提供了用于[迁移](#)的轻型替代方法，用于管理数据库架构。这些方法在以下情况下很有用：数据是暂时性的，可以在架构更改时删除。例如在原型制作、测试期间或用于本地缓存。

某些提供程序（尤其是非关系的）不支持迁移。对于这些提供程序，EnsureCreated 通常是初始化数据库架构的最简单方法。

WARNING

EnsureCreated 和迁移不能很好地协同工作。如果使用了“迁移”，则不要使用 EnsureCreated 来初始化架构。

从 EnsureCreated 到“迁移”的过渡不是无缝体验。要执行此操作，最简单的方法是删除数据库，然后使用迁移重新创建数据库。如果预计将来使用迁移，最好只开始迁移，而不是使用 EnsureCreated。

EnsureDeleted

如果数据库存在，EnsureDeleted 方法会将其删除。如果你没有相应的权限，则会引发异常。

```
// Drop the database if it exists  
dbContext.Database.EnsureDeleted();
```

EnsureCreated

如果数据库不存在，EnsureCreated 将创建该数据库，并初始化该数据库的架构。如果存在任何表（包括另一个 DbContext 类的表），则不会初始化架构。

```
// Create the database if it doesn't exist  
dbContext.Database.EnsureCreated();
```

TIP

这些方法的异步版本也可用。

SQL 脚本

若要获取 EnsureCreated 使用的 SQL，可以使用 GenerateCreateScript 方法。

```
var sql = dbContext.Database.GenerateCreateScript();
```

多个 DbContext 类

仅当数据库中不存在表格时，EnsureCreated 才会起作用。如果有必要，也可以编写自己的检查，来查看架构是否需要进行初始化，并可通过基础 [IRelationalDatabaseCreator](#) 服务来初始化架构。

```
// TODO: Check whether the schema needs to be initialized  
  
// Initialize the schema for this DbContext  
var databaseCreator = dbContext.GetService<IRelationalDatabaseCreator>();  
databaseCreator.CreateTables();
```

反向工程

2020/3/11 •

反向工程是基架实体类型类的过程，以及基于数据库架构的 DbContext 类。可以使用 EF Core 程序包管理器控制台 (PMC) 工具的 `Scaffold-DbContext` 命令或 .NET 命令行接口 (CLI) 工具的 `dotnet ef dbcontext scaffold` 命令来执行该命令。

安装

在进行反向工程之前，你需要安装[PMC 工具](#)(仅适用于 Visual Studio)或[CLI 工具](#)。有关详细信息，请参阅链接。

还需要为要进行反向工程的数据库架构安装适当的[数据库提供程序](#)。

连接字符串

该命令的第一个参数是指向数据库的连接字符串。工具将使用此连接字符串来读取数据库架构。

引号和转义连接字符串的方式取决于您使用哪个 shell 执行命令。有关详细信息，请参阅 shell 的文档。例如，PowerShell 需要转义 `$` 字符，但不能 `\`。

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook'  
Microsoft.EntityFrameworkCore.SqlServer
```

```
dotnet ef dbcontext scaffold "Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook"  
Microsoft.EntityFrameworkCore.SqlServer
```

配置和用户机密

如果有 ASP.NET Core 项目，则可以使用 `Name=<connection-string>` 语法从配置中读取连接字符串。

这很适合用于[机密管理器工具](#)，使数据库密码与代码库保持分离。

```
dotnet user-secrets set ConnectionStrings.Chinook "Data Source=(localdb)\MSSQLLocalDB;Initial  
Catalog=Chinook"  
dotnet ef dbcontext scaffold Name=Chinook Microsoft.EntityFrameworkCore.SqlServer
```

提供程序名称

第二个参数是提供程序名称。提供程序名称通常与提供程序的 NuGet 包名称相同。

指定表

默认情况下，将数据库架构中的所有表反向工程为实体类型。您可以通过指定架构和表来限制对哪些表进行反向工程。

PMC 中的 `-Schemas` 参数和 CLI 中的 `--schema` 选项可用于包含架构中的每个表。

`-Tables` (PMC) 和 `--table` (CLI) 可用于包含特定表。

若要在 PMC 中包括多个表，请使用数组。

```
Scaffold-DbContext ... -Tables Artist, Album
```

若要在 CLI 中包含多个表, 请多次指定选项。

```
dotnet ef dbcontext scaffold ... --table Artist --table Album
```

保留名称

默认情况下, 表和列名已固定, 以便更好地匹配类型和属性的 .NET 命名约定。在 PMC 中指定 `-UseDatabaseNames` 开关或在 CLI 中指定 `--use-database-names` 选项将禁用这一行为, 从而尽可能保留原始数据库名称。无效的 .NET 标识符仍然是固定的, 而合成的名称(如导航属性)仍符合 .NET 命名约定。

熟知 API 或数据批注

默认情况下, 实体类型是使用熟知 API 配置的。将 `-DataAnnotations` (PMC) 或 `--data-annotations` (CLI) 指定为尽可能使用数据批注。

例如, 使用熟知的 API 将基架:

```
entity.Property(e => e.Title)
    .IsRequired()
    .HasMaxLength(160);
```

使用数据注释时, 将基架:

```
[Required]
[StringLength(160)]
public string Title { get; set; }
```

DbContext 名称

默认情况下, 基架 DbContext 类名称将以默认值作为后缀的数据库的名称。若要指定其他帐户, 请使用 PMC 中的 `-Context`, 并在 CLI 中 `--context`。

目录和命名空间

实体类和 DbContext 类将基架到项目的根目录中, 并使用项目的默认命名空间。可以使用 `-OutputDir` (PMC) 或 `--output-dir` (CLI) 指定基架的目录。命名空间将为根命名空间加上项目根目录下的任何子目录的名称。

你还可以使用 `-ContextDir` (PMC) 和 `--context-dir` (CLI) 将 DbContext 类基架到不同于实体类型类的目录中。

```
Scaffold-DbContext ... -ContextDir Data -OutputDir Models
```

```
dotnet ef dbcontext scaffold ... --context-dir Data --output-dir Models
```

工作原理

反向工程从读取数据库架构开始。它读取有关表、列、约束和索引的信息。

接下来, 它使用架构信息创建 EF Core 模型。表用于创建实体类型, 列用于创建属性, 和外键用于创建关系。

最后，模型用于生成代码。为了从应用程序重新创建相同的模型，相应的实体类型类、熟知 API 和数据批注都是基架的。

限制

- 不是有关模型的所有内容都可以使用数据库架构来表示。例如，有关[继承层次结构、附属类型和表拆分](#)的信息在数据库架构中不存在。因此，这些构造永远不会经过反向工程。
- 此外，EF Core 提供程序可能不支持某些列类型。这些列不会包含在模型中。
- 可以在 EF Core 模型中定义[并发标记](#)，以防止两个用户同时更新同一实体。某些数据库有一种特殊类型的类型来表示此类型的列（例如 SQL Server 中的 rowversion），在这种情况下，我们可以对此信息进行反向工程。但是，其他并发令牌不会进行反向工程。
- [反向C#工程当前不支持可为 null 的引用类型功能](#)：EF Core 始终会生成假定禁用该功能的代码。例如，可为 null 的文本列将被基架为具有类型 `string` 的属性，而不是 `string?`，其中使用的是用于配置是否需要属性的流畅 API 或数据批注。您可以编辑基架代码并将其替换为 C# 可为 null 的批注。[#15520](#) 的问题跟踪了可为 null 的引用类型的基架支持。

自定义模型

EF Core 生成的代码是您的代码。随意更改。仅当您再次对同一模型进行反向工程时，才会重新生成它。基架代码表示一个可用于访问数据库的模型，但它当然不是唯一可以使用的模型。

根据需要自定义实体类型类和 DbContext 类。例如，你可以选择重命名类型和属性、引入继承层次结构或将表拆分为多个实体。还可以从模型中删除非唯一索引、未使用的序列和导航属性、可选的标量属性和约束名称。

还可以添加其他构造函数、方法、属性等。在单独的文件中使用另一个分部类。即使您要再次对模型进行反向工程，此方法也能正常工作。

更新模型

更改数据库后，可能需要更新 EF Core 模型以反映这些更改。如果数据库更改很简单，只需手动对 EF Core 模型进行更改即可。例如，对表或列进行重命名、删除列或更新列的类型是在代码中进行的一些简单的更改。

但是，更重要的更改不容易手动完成。一个常见的工作流是通过使用 `-Force` (PMC) 或 `--force` (CLI)，使用更新的模型覆盖现有模型，从数据库反向对模型进行反向工程。

另一个常请求的功能是能够从数据库更新模型，同时保留自定义项（如重命名、类型层次结构等）。使用问题[#831](#) 跟踪此功能的进度。

WARNING

如果您从数据库中再次对模型进行反向工程，则对这些文件所做的任何更改都将丢失。

查询数据

2020/4/8 • [Edit Online](#)

Entity Framework Core 使用语言集成查询 (LINQ) 来查询数据库中的数据。通过 LINQ 可使用 C#(或你选择的其他 .NET 语言)编写强类型查询。它使用你派生得到的上下文和实体类来引用数据库对象。EF Core 将 LINQ 查询的表示形式传递给数据库提供程序。反过来，数据库提供程序将其转换为数据库特定的查询语言(例如，用于关系数据库的 SQL)。

TIP

可在 GitHub 上查看此文章的示例。

以下片段显示的几个示例展示了如何使用 Entity Framework Core 完成常见任务。

加载所有数据

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.ToList();
}
```

加载单个实体

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);
}
```

Filtering

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

延伸阅读

- 了解有关 [LINQ 查询表达式](#) 的详细信息
- 要更详细地了解如何在 EF Core 中处理查询，请参阅[查询的工作原理](#)。

客户端与服务器评估

2020/4/8 • [Edit Online](#)

作为一般规则，Entity Framework Core 会尝试尽可能全面地评估服务器上的查询。EF Core 将查询的一部分转换为可在客户端评估的参数。系统将查询的其余部分（及生成的参数）提供给数据库提供程序，以确定要在服务器上评估的等效数据库查询。EF Core 支持在顶级投影中进行部分客户端评估（基本上为最后一次调用 `Select()`）。如果查询中的顶级投影无法转换为服务器，EF Core 将从服务器中提取任何所需的数据，并在客户端上评估查询的其余部分。如果 EF Core 在顶级投影之外的任何位置检测到不能转换为服务器的表达式，则会引发运行时异常。请参阅[查询工作原理](#)，了解 EF Core 如何确定哪些表达式无法转换为服务器。

NOTE

在 3.0 版之前，Entity Framework Core 支持在查询中的任何位置进行客户端评估。有关详细信息，请参阅[历史版本部分](#)。

TIP

可在 GitHub 上查看此文章的[示例](#)。

顶级投影中的客户端评估

在下面的示例中，一个辅助方法用于标准化从 SQL Server 数据库中返回的博客的 URL。由于 SQL Server 提供程序不了解此方法的实现方式，因此无法将其转换为 SQL。查询的所有其余部分是在数据库中评估的，但通过此方法传递返回的 `URL` 却是在客户端上完成。

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(blog => new
    {
        Id = blog.BlogId,
        Url = StandardizeUrl(blog.Url)
    })
    .ToList();
```

```
public static string StandardizeUrl(string url)
{
    url = url.ToLower();

    if (!url.StartsWith("http://"))
    {
        url = string.Concat("http://", url);
    }

    return url;
}
```

不支持的客户端评估

尽管客户端评估非常有用，但有时会减弱性能。请看以下查询，其中的 `where` 筛选器现已使用辅助方法。由于数据库中不能应用筛选器，因此需要将所有数据提取到内存中，以便在客户端上应用筛选器。根据服务器上的筛选器和数据量，客户端评估可能会减弱性能。因此 Entity Framework Core 会阻止此类客户端评估，并引发运行时异常。

```
var blogs = context.Blogs
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

显式客户端评估

在某些情况下，可能需要以显式方式强制进行客户端评估，如下所示

- 由于数据量小，因此在进行客户端评估时才不会大幅减弱性能。
- 所用的 LINQ 运算符不会进行任何服务器端转换。

在这种情况下，通过调用 `AsEnumerable` 或 `ToList` 等方法（若为异步，则调用 `AsAsyncEnumerable` 或 `ToListAsync`），以显式方式选择进行客户端评估。使用 `AsEnumerable` 将对结果进行流式传输，但使用 `ToList` 将通过创建列表来进行缓冲，因此也会占用额外的内存。但如果枚举多次，则将结果存储到列表中可以带来更大的帮助，因为只有一个对数据库的查询。根据具体的使用情况，你应该评估哪种方法更适合。

```
var blogs = context.Blogs
    .AsEnumerable()
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

客户端评估中潜在的内存泄漏

由于查询转换和编译的开销高昂，因此 EF Core 会缓存已编译的查询计划。缓存的委托在对顶级投影进行客户端评估时可能会使用客户端代码。EF Core 为树型结构中客户端评估的部分生成参数，并通过替换参数值重用查询计划。但表达式树中的某些常数无法转换为参数。如果缓存的委托包含此类常数，则无法将这些对象垃圾回收，因为它们仍被引用。如果此类对象包含 `DbContext` 或其中的其他服务，则会导致应用的内存使用量逐渐增多。此行为通常是内存泄漏的标志。只要遇到的常数为不能使用当前数据库提供程序映射的类型，EF Core 就会引发异常。常见原因及其解决方案如下所示：

- 使用实例方法**：在客户端投影中使用实例方法时，表达式树包含实例的常数。如果你的方法不使用该实例中的任何数据，请考虑将该方法设为静态方法。如果需要方法主体中的实例数据，则将特定数据作为实参传递给方法。
- 将常数实参传递给方法**：这种情况通常是由于在客户端方法的实参中使用 `this` 引起的。请考虑将实参拆分为多个标量实参，可由数据库提供程序进行映射。
- 其他常数**：如果在任何其他情况下都出现常数，则可以评估在处理过程中是否需要该常数。如果必须具有常数，或者如果无法使用上述情况中的解决方案，则创建本地变量来存储值，并在查询中使用局部变量。EF Core 会将局部变量转换为形参。

早期版本

以下部分适用于 3.0 以前的 EF Core 版本。

旧的 EF Core 版本支持在查询的任何部分中进行客户端评估，而不仅仅是顶级投影。因此，与[不支持的客户评估](#)部分下发布的查询类似的查询可以正常工作。由于此行为可能引起不易觉察的性能问题，EF Core 记录了客户端评估警告。有关如何查看日志记录输出的详细信息，请参阅[日志记录](#)。

（可选）借助 EF Core，你可以将默认行为更改为在执行客户端评估时引发异常或不执行任何操作（在投影中除外）。引发异常的行为会使其类似于 3.0 中的行为。若要更改该行为，你需要在设置上下文选项时配置警告。上下文选项一般在 `DbContext.OnConfiguring` 中设置，如果使用 ASP.NET Core，则在 `Startup.cs` 中设置。

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EFQuerying;Trusted_Connection=True;")
        .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
}
```

跟踪与非跟踪查询

2020/4/8 • [Edit Online](#)

跟踪行为决定了 Entity Framework Core 是否将有关实体实例的信息保留在其更改跟踪器中。如果已跟踪某个实体，则该实体中检测到的任何更改都会在 `SaveChanges()` 期间永久保存到数据库。EF Core 还将修复跟踪查询结果中的实体与更改跟踪器中的实体之间的导航属性。

NOTE

从不跟踪无键实体类型。无论在何处提到实体类型，它都是指定义了键的实体类型。

TIP

可在 GitHub 上查看此文章的示例。

跟踪查询

返回实体类型的查询是默认会被跟踪的。这表示可以更改这些实体实例，然后通过 `SaveChanges()` 持久化这些更改。在以下示例中，将检测到对博客评分所做的更改，并在 `SaveChanges()` 期间将这些更改持久化到数据库中。

```
var blog = context.Blogs.SingleOrDefault(b => b.BlogId == 1);
blog.Rating = 5;
context.SaveChanges();
```

非跟踪查询

在只读方案中使用结果时，非跟踪查询十分有用。可以更快速地执行非跟踪查询，因为无需设置更改跟踪信息。如果不需要更新从数据库中检索到的实体，则应使用非跟踪查询。可以将单个查询替换为非跟踪查询。

```
var blogs = context.Blogs
    .AsNoTracking()
    .ToList();
```

还可以在上下文实例级别更改默认跟踪行为：

```
context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

var blogs = context.Blogs.ToList();
```

标识解析

由于跟踪查询使用更改跟踪器，因此 EF Core 将在跟踪查询中执行标识解析。当具体化实体时，如果 EF Core 已被跟踪，则会从更改跟踪器返回相同的实体实例。如果结果中多次包含相同的实体，则每次会返回相同的实例。非跟踪查询不会使用更改跟踪器，也不会执行标识解析。因此会返回实体的新实例，即使结果中多次包含相同的实体也是如此。此行为与 EF Core 3.0 之前版本中的行为有所不同，请参阅[早期版本](#)。

跟踪和自定义投影

即使查询的结果类型不是实体类型，默认情况下 EF Core 也会跟踪结果中包含的实体类型。在以下返回匿名类型的查询中，结果集中的 `Blog` 实例会被跟踪。

```
var blog = context.Blogs
    .Select(b =>
        new
        {
            Blog = b,
            PostCount = b.Posts.Count()
        });
    
```

如果结果集包含来自 LINQ 组合的实体类型，EF Core 将跟踪它们。

```
var blog = context.Blogs
    .Select(b =>
        new
        {
            Blog = b,
            Post = b.Posts.OrderBy(p => p.Rating).LastOrDefault()
        });
    
```

如果结果集不包含任何实体类型，则不会执行跟踪。在以下查询中，我们返回匿名类型（具有实体中的某些值，但没有实际实体类型的实例）。查询中没有任何被跟踪的实体。

```
var blog = context.Blogs
    .Select(b =>
        new
        {
            Id = b.BlogId,
            Url = b.Url
        });
    
```

EF Core 支持执行顶级投影中的客户端评估。如果 EF Core 具体化实体实例以进行客户端评估，则会跟踪该实体实例。此处，由于我们要将 `blog` 实体传递到客户端方法 `StandardizeURL`，因此 EF Core 也会跟踪博客实例。

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(blog => new
    {
        Id = blog.BlogId,
        Url = StandardizeUrl(blog)
    })
    .ToList();
    
```

```
public static string StandardizeUrl(Blog blog)
{
    var url = blog.Url.ToLower();

    if (!url.StartsWith("http://"))
    {
        url = string.Concat("http://", url);
    }

    return url;
}
    
```

EF Core 不会跟踪结果中包含的无键实体实例。但 EF Core 会根据上述规则跟踪带有键的实体类型的所有其他实例。

在 EF Core 3.0 之前，某些上述规则的工作方式有所不同。有关详细信息，请参阅[早期版本](#)。

早期版本

在 3.0 版之前，EF Core 执行跟踪的方式有一些差异。显著的差异如下：

- 如[客户端与服务器评估](#)页中所述，在 3.0 版之前，EF Core 支持在查询的任何部分中执行客户端评估。客户端评估导致了实体的具体化，这不是结果的一部分。因此 EF Core 分析了结果以检测要跟踪的内容。此设计有一些不同之处，如下所示：
 - 投影中的客户端评估（导致具体化，但未返回具体化的实体实例）未被跟踪。以下示例未跟踪 `blog` 实体。

```
var blogs = context.Blogs
    .OrderByDescending(blog => blog.Rating)
    .Select(blog => new
{
    Id = blog.BlogId,
    Url = StandardizeUrl(blog)
})
.ToList();
```

- 在某些情况下，EF Core 未跟踪来自 LINQ 组合的对象。以下示例未跟踪 `Post`。

```
var blog = context.Blogs
    .Select(b =>
        new
        {
            Blog = b,
            Post = b.Posts.OrderBy(p => p.Rating).LastOrDefault()
        });

```

- 只要查询结果中包含无键实体类型，整个查询就会进行非跟踪。这表示不会跟踪结果中包含的带有键的实体类型。
- EF Core 在非跟踪查询中执行标识解析。它使用了弱引用来跟踪已返回的实体。因此，如果结果集多次包含相同的实体，则每次会返回相同的实例。尽管具有相同标识的上一个结果超出了范围并进行了垃圾回收，EF Core 也会返回新实例。

复杂查询运算符

2020/4/8 • [Edit Online](#)

语言集成查询 (LINQ) 包含许多用于组合多个数据源或执行复杂处理的复杂运算符。并非所有 LINQ 运算符都会在服务器端进行适当转换。有时，采用一种形式的查询会转换为服务器，但如果采用另一种形式，即使结果相同，也不会转换。本页介绍部分复杂运算符及其支持的变体。在将来的版本中，我们可能会认识更多的模式并添加其相应的转换。另请牢记，转换支持在提供程序之间各所不同。在 SqlServer 中转换的特定查询可能无法用于 SQLite 数据库。

TIP

可在 GitHub 上查看此文章的示例。

Join

借助 LINQ Join 运算符，可根据每个源的键选择器连接两个数据源，并在键匹配时生成值的元组。该运算符在关系数据库中自然而然地转换为 `INNER JOIN`。虽然 LINQ Join 具有外部和内部键选择器，但数据库只需要一个联接条件。因此 EF Core 通过比较外部键选择器和内部键选择器是否相等，来生成联接条件。此外，如果键选择器是匿名类型，则 EF Core 会生成一个联接条件以用于比较组件是否相等。

```
var query = from photo in context.Set<PersonPhoto>()
            join person in context.Set<Person>()
            on photo.PersonPhotoId equals person.PhotoId
            select new { person, photo };
```

```
SELECT [p].[PersonId], [p].[Name], [p].[PhotoId], [p0].[PersonPhotoId], [p0].[Caption], [p0].[Photo]
FROM [PersonPhoto] AS [p0]
INNER JOIN [Person] AS [p] ON [p0].[PersonPhotoId] = [p].[PhotoId]
```

GroupJoin

借助 LINQ GroupJoin 运算符，可以采用与 Join 类似的方式连接两个数据源，但它会创建一组内部值，用于匹配外部元素。执行与以下示例类似的查询将生成 `Blog` & `IEnumerable<Post>` 的结果。由于数据库(特别是关系数据库)无法表示一组客户端对象，因此在许多情况下，GroupJoin 不会转换为服务器。它需要从服务器获取所有数据来进行 GroupJoin，无需使用特殊选择器(下面的第一个查询)。但如果选择器限制选定的数据，则从服务器提取所有数据可能会导致出现性能问题(下面的第二个查询)。这就是 EF Core 不转换 GroupJoin 的原因。

```
var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
            on b.BlogId equals p.PostId into grouping
            select new { b, grouping };
```

```
var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
            on b.BlogId equals p.PostId into grouping
            select new { b, Posts = grouping.Where(p => p.Content.Contains("EF")).ToList() };
```

SelectMany

借助 LINQ SelectMany 运算符，可为每个外部元素枚举集合选择器，并从每个数据源生成值的元组。在某种程度上，它是没有任何条件的联接，因此每个外部元素都与来自集合源的元素连接。根据集合选择器与外部数据源的关系，SelectMany 可在服务器端转换为各种不同的查询。

集合选择器不引用外部

如果集合选择器不引用外部源中的任何内容，则结果是这两个数据源的笛卡尔乘积。它在关系数据库中转换为

CROSS JOIN。

```
var query = from b in context.Set<Blog>()
            from p in context.Set<Post>()
            select new { b, p };
```

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
CROSS JOIN [Posts] AS [p]
```

集合选择器引用 where 子句中的外部

如果集合选择器具有引用外部元素的 where 子句，则 EF Core 会将其转换为数据库联接，并将谓词用作联接条件。在对外部元素使用集合导航作为集合选择器时，通常会出现这种情况。如果外部元素的集合为空，则不会为该外部元素生成任何结果。但如果在集合选择器上应用 DefaultIfEmpty，则外部元素将与内部元素的默认值连接。由于这种区别，应用 INNER JOIN 时，如果缺少 DefaultIfEmpty 和 LEFT JOIN，此类查询会转换为 DefaultIfEmpty。

```
var query = from b in context.Set<Blog>()
            from p in context.Set<Post>().Where(p => b.BlogId == p.BlogId)
            select new { b, p };

var query2 = from b in context.Set<Blog>()
            from p in context.Set<Post>().Where(p => b.BlogId == p.BlogId).DefaultIfEmpty()
            select new { b, p };
```

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
INNER JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]
```

集合选择器引用非 where 情况下的外部

如果集合选择器引用的外部元素不在 where 子句中（如上所述），则它不会转换为数据库联接。这就是我们需要为每个外部元素评估集合选择器的原因。它在许多关系数据库中转换为 APPLY 操作。如果外部元素的集合为空，则不会为该外部元素生成任何结果。但如果在集合选择器上应用 DefaultIfEmpty，则外部元素将与内部元素的默认值连接。由于这种区别，应用 CROSS APPLY 时，如果缺少 DefaultIfEmpty 和 OUTER APPLY，此类查询会转换为 DefaultIfEmpty。像 SQLite 之类的某些数据库不支持 APPLY 运算符，因此此类查询可能不会进行转换。

```

var query = from b in context.Set<Blog>()
            from p in context.Set<Post>().Select(p => b.Url + "=" + p.Title)
            select new { b, p };

var query2 = from b in context.Set<Blog>()
            from p in context.Set<Post>().Select(p => b.Url + "=" + p.Title).DefaultIfEmpty()
            select new { b, p };

```

```

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], ([b].[Url] + N'=>') + [p].[Title] AS [p]
FROM [Blogs] AS [b]
CROSS APPLY [Posts] AS [p]

```

```

SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], ([b].[Url] + N'=>') + [p].[Title] AS [p]
FROM [Blogs] AS [b]
OUTER APPLY [Posts] AS [p]

```

GroupBy

LINQ GroupBy 运算符创建 `IGrouping< TKey, TElement >` 类型的结果，其中 `TKey` 和 `TElement` 可以是任意类型。此外，`IGrouping` 实现了 `IEnumerable< TElement >`，这意味着可在分组后使用任意 LINQ 运算符来对其进行组合。由于任何数据库结构都无法表示 `IGrouping`，因此在大多数情况下 GroupBy 运算符不会进行任何转换。聚合运算符应用于返回标量的每个组时，该运算符可在关系数据库中转换为 SQL `GROUP BY`。SQL `GROUP BY` 也会受到限制。它要求只按标量值进行分组。投影只能包含分组键列或对列应用的任何聚合。EF Core 标识此模式并将其转换为服务器，如下示例中所示：

```

var query = from p in context.Set<Post>()
            group p by p.AuthorId into g
            select new
            {
                g.Key,
                Count = g.Count()
            };

```

```

SELECT [p].[AuthorId] AS [Key], COUNT(*) AS [Count]
FROM [Posts] AS [p]
GROUP BY [p].[AuthorId]

```

EF Core 还会转换符合以下条件的查询：分组的聚合运算符出现在 Where 或 OrderBy(或其他排序方式) LINQ 运算符中。它在 SQL 中将 `HAVING` 子句用于 where 子句。在应用 GroupBy 运算符之前的查询部分可以是任何复杂查询，只要它可转换为服务器即可。此外，将聚合运算符应用于分组查询以从生成的源中移除分组后，可以像使用任何其他查询一样，在它的基础上进行组合。

```

var query = from p in context.Set<Post>()
            group p by p.AuthorId into g
            where g.Count() > 0
            orderby g.Key
            select new
            {
                g.Key,
                Count = g.Count()
            };

```

```
SELECT [p].[AuthorId] AS [Key], COUNT(*) AS [Count]
FROM [Posts] AS [p]
GROUP BY [p].[AuthorId]
HAVING COUNT(*) > 0
ORDER BY [p].[AuthorId]
```

EF Core 支持的聚合运算符如下所示

- 平均值
- Count
- LongCount
- Max
- Min
- SUM

Left Join

虽然 Left Join 不是 LINQ 运算符，但关系数据库具有常用于查询的 Left Join 的概念。LINQ 查询中的特定模式提供与服务器上的 `LEFT JOIN` 相同的结果。EF Core 标识此类模式，并在服务器端生成等效的 `LEFT JOIN`。该模式包括在两个数据源之间创建 GroupJoin，然后通过对分组源使用 SelectMany 运算符与 DefaultIfEmpty 来平展分组，从而在内部不具有相关元素时匹配 null。下面的示例显示该模式的样式及其生成的内容。

```
var query = from b in context.Set<Blog>()
            join p in context.Set<Post>()
                on b.BlogId equals p.BlogId into grouping
            from p in grouping.DefaultIfEmpty()
            select new { b, p };
```

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url], [p].[PostId], [p].[AuthorId], [p].[BlogId], [p].[Content], [p].[Rating], [p].[Title]
FROM [Blogs] AS [b]
LEFT JOIN [Posts] AS [p] ON [b].[BlogId] = [p].[BlogId]
```

以上模式在表达式树中创建复杂的结构。因此，EF Core 要求在紧随运算符的步骤中将 GroupJoin 运算符的分组结果平展。即使使用了 GroupJoin-DefaultIfEmpty-SelectMany，但采用其他的模式，也不能将其标识为 Left Join。

加载相关数据

2020/4/8 • [Edit Online](#)

Entity Framework Core 允许在模型中使用导航属性来加载关联实体。有三种常见的 O/RM 模式可用于加载关联数据。

- **预先加载**表示从数据库中加载关联数据，作为初始查询的一部分。
- **显式加载**表示稍后从数据库中显式加载关联数据。
- **延迟加载**表示在访问导航属性时，从数据库中以透明方式加载关联数据。

TIP

可在 GitHub 上查看此文章的[示例](#)。

预先加载

可以使用 `Include` 方法来指定要包含在查询结果中的关联数据。在以下示例中，结果中返回的 `blogs` 将使用关联的 `posts` 填充其 `Posts` 属性。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

TIP

Entity Framework Core 会根据之前已加载到上下文实例中的实体自动填充导航属性。因此，即使不显式包含导航属性的数据，如果先前加载了部分或所有关联实体，则仍可能填充该属性。

可以在单个查询中包含多个关系的关联数据。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
}
```

包含多个层级

使用 `ThenInclude` 方法可以依循关系包含多个层级的关联数据。以下示例加载了所有博客、其关联文章及每篇文章的作者。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
    .ToList();
}
```

可通过链式调用 `ThenInclude`，进一步包含更深级别的关联数据。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
    .ToList();
}
```

可以将来自多个级别和多个根的关联数据合并到同一查询中。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
        .Include(blog => blog.Owner)
        .ThenInclude(owner => owner.Photo)
    .ToList();
}
```

你可能希望将已包含的某个实体的多个关联实体都包含进来。例如，当查询 `Blogs` 时，你会包含 `Posts`，然后希望同时包含 `Author` 的 `Tags` 和 `Posts`。为此，需要从根级别开始指定每个包含路径。例如，`Blog -> Posts -> Author` 和 `Blog -> Posts -> Tags`。这并不意味着会获得冗余联接查询，在大多数情况下，EF 会在生成 SQL 时合并相应的联接查询。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Tags)
    .ToList();
}
```

Caution

从版本 3.0.0 开始，每个 `Include` 都将导致向关系提供程序生成的 SQL 查询添加额外的 JOIN，而以前的版本则生成其他 SQL 查询。这可以显著地改变(提升或降低)查询性能。具体而言，具有大量 `Include` 运算符的 LINQ 查询可能需要将分解为多个单独的 LINQ 查询，以避免笛卡尔爆炸问题。

派生类型上的包含

可以使用 `Include` 和 `ThenInclude` 包含仅在派生类型上定义的导航的关联数据。

给定以下模型：

```
public class SchoolContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<School> Schools { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<School>().HasMany(s => s.Students).WithOne(s => s.School);
    }
}

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Student : Person
{
    public School School { get; set; }
}

public class School
{
    public int Id { get; set; }
    public string Name { get; set; }

    public List<Student> Students { get; set; }
}
```

对于具有学生身份的所有人员，可使用多种方法来预先加载其 `School` 导航属性的内容：

- 使用强制转换

```
context.People.Include(person => ((Student)person).School).ToList()
```

- 使用 `as` 运算符

```
context.People.Include(person => (person as Student).School).ToList()
```

- 使用 `Include` 的重载，该方法采用 `string` 类型的参数

```
context.People.Include("School").ToList()
```

显式加载

可以通过 `DbContext.Entry(...)` API 显式加载导航属性。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    context.Entry(blog)
        .Collection(b => b.Posts)
        .Load();

    context.Entry(blog)
        .Reference(b => b.Owner)
        .Load();
}
```

还可以通过执行返回关联实体的单独查询来显式加载导航属性。如果已启用更改跟踪，则在加载实体时，EF Core 将自动设置新加载的实体的导航属性以引用任何已加载的实体，并设置已加载实体的导航属性以引用新加载的实体。

查询关联实体

还可以获得表示导航属性内容的 LINQ 查询。

这样就能做到(譬如)无需将关联实体加载到内存中，就可以对关联实体执行聚合运算。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

还可以筛选要加载到内存中的关联实体。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Single(b => b.BlogId == 1);

    var goodPosts = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Rating > 3)
        .ToList();
}
```

延迟加载

使用延迟加载的最简单方式是通过安装 [Microsoft.EntityFrameworkCore.Proxies](#) 包，并通过调用 `UseLazyLoadingProxies` 来启用该包。例如：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
        .UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString);
```

或在使用 AddDbContext 时：

```
.AddDbContext<BlogginContext>(
    b => b.UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString));
```

EF Core 接着会为可重写的任何导航属性(即, 必须是 `virtual` 且在可被继承的类上)启用延迟加载。例如, 在以下实体中, `Post.Blog` 和 `Blog.Posts` 导航属性将被延迟加载。

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

    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public virtual Blog Blog { get; set; }
}
```

不使用代理的延迟加载

使用代理进行延迟加载的工作方式是将 `ILazyLoader` 注入到实体中, 如[实体类型构造函数](#)中所述。例如:

```

public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(ILazyLoader lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private ILazyLoader LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}

```

这不要求实体类型为可继承的类型，也不要求导航属性必须是虚拟的，且允许通过 `new` 创建的实体实例在附加到上下文后可进行延迟加载。但它需要对 `ILazyLoader` Microsoft.EntityFrameworkCore.Abstractions 包中定义的服务的引用。此包包含所允许的最少的一组类型，以便将依赖此包时所产生的影响降至最低。不过，可以将 `ILazyLoader.Load` 方法以委托的形式注入，这样就可以完全避免依赖于实体类型的任何 EF Core 包。例如：

```
public class Blog
{
    private ICollection<Post> _posts;

    public Blog()
    {
    }

    private Blog(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Post> Posts
    {
        get => LazyLoader.Load(this, ref _posts);
        set => _posts = value;
    }
}

public class Post
{
    private Blog _blog;

    public Post()
    {
    }

    private Post(Action<object, string> lazyLoader)
    {
        LazyLoader = lazyLoader;
    }

    private Action<object, string> LazyLoader { get; set; }

    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog
    {
        get => LazyLoader.Load(this, ref _blog);
        set => _blog = value;
    }
}
```

上述代码使用 `Load` 扩展方法，以便更干净地使用委托：

```
public static class PocoLoadingExtensions
{
    public static TRelated Load<TRelated>(
        this Action<object, string> loader,
        object entity,
        ref TRelated navigationField,
        [CallerMemberName] string navigationName = null)
        where TRelated : class
    {
        loader?.Invoke(entity, navigationName);

        return navigationField;
    }
}
```

NOTE

延迟加载委托的构造函数参数必须名为“lazyLoader”。未来的一个版本中的配置将计划采用另一个名称。

关联数据和序列化

由于 EF Core 会自动修正导航属性，因此在对象图中可能会产生循环引用。例如，加载博客及其关联文章会生成引用文章集合的博客对象。而其中每篇文章又会引用该博客。

某些序列化框架不允许使用循环引用。例如，Json.NET 在产生循环引用的情况下，会引发以下异常。

```
Newtonsoft.Json.JsonSerializationException: 为“MyApplication.Models.Blog”类型的“Blog”属性检测到自引用循环。
```

如果正在使用 ASP.NET Core，则可以将 Json.NET 配置为忽略在对象图中找到的循环引用。这是在 `ConfigureServices(...)` 中通过 `Startup.cs` 方法实现的。

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddMvc()
        .AddJsonOptions(
            options => options.SerializerSettings.ReferenceLoopHandling =
Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );

    ...
}
```

另一种方法是使用 `[JsonIgnore]` 特性修饰其中一个导航属性，该特性指示 Json.NET 在序列化时不遍历该导航属性。

异步查询

2020/4/8 • [Edit Online](#)

当在数据库中执行查询时，异步查询可避免阻止线程。异步查询对于在胖客户端应用程序中保持响应式 UI 非常重要。异步查询还可以增加 Web 应用程序中的吞吐量，即通过释放线程，以处理 Web 应用程序中的其他请求。有关详细信息，请参阅[使用 C# 异步编程](#)。

WARNING

EF Core 不支持在同一上下文实例上运行多个并行操作。应始终等待操作完成，然后再开始下一个操作。这通常是通过在每个异步操作上使用 `await` 关键字完成的。

Entity Framework Core 提供一组类似于 LINQ 方法的异步扩展方法，用于执行查询并返回结果。示例包括

`ToListAsync()`、`ToDictionaryAsync()`、`SingleAsync()`。某些 LINQ 运算符（如 `Where(...)` 或 `OrderBy(...)`）没有对应的异步版本，因为这些方法仅用于构建 LINQ 表达式树，而不会导致在数据库中执行查询。

IMPORTANT

EF Core 异步扩展方法在 `Microsoft.EntityFrameworkCore` 命名空间中定义。必须导入此命名空间才能使这些方法可用。

```
public async Task<List<Blog>> GetBlogsAsync()
{
    using (var context = new BloggingContext())
    {
        return await context.Blogs.ToListAsync();
    }
}
```

原生 SQL 查询

2020/4/8 • [Edit Online](#)

通过 Entity Framework Core 可以在使用关系数据库时下降到原始 SQL 查询。所需查询不能使用 LINQ 来表示时，可以使用原始 SQL 查询。如果使用 LINQ 查询导致 SQL 查询效率低下，也可以使用原始 SQL 查询。原始 SQL 查询可返回一般实体类型或者模型中的无键实体类型。

TIP

可在 GitHub 上查看此文章的示例。

基本原生 SQL 查询

可使用 `FromSqlRaw` 扩展方法基于原始 SQL 查询开始 LINQ 查询。`FromSqlRaw` 只能在直接位于 `DbSet<>` 上的查询根上使用。

```
var blogs = context.Blogs
    .FromSqlRaw("SELECT * FROM dbo.Blogs")
    .ToList();
```

原生 SQL 查询可用于执行存储过程。

```
var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

传递参数

WARNING

■ SQL ■

向原始 SQL 查询引入任何用户提供的值时，必须注意防范 SQL 注入攻击。除了验证确保此类值不包含无效字符，请始终使用会将值与 SQL 文本分开发送的参数化处理。

具体而言，如果连接和内插的字符串 (`""`) 带有用户提供的未经验证的值，则切勿将其传递到 `FromSqlRaw` 或 `ExecuteSqlRaw`。通过 `FromSqlInterpolated` 和 `ExecuteSqlInterpolated` 方法，可采用一种能抵御 SQL 注入攻击的方式使用字符串内插语法。

下面的示例通过在 SQL 查询字符串中包含形参占位符并提供额外的实参，将单个形参传递到存储过程。虽然此语法可能看上去像 `String.Format` 语法，但提供的值包装在 `SqlParameter` 中，且生成的参数名称插入到指定 `{0}` 占位符的位置。

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

虽然 `FromSqlInterpolated` 类似于 `FromSqlRaw`，但你可以借助它使用字符串内插语法。与 `FromSqlRaw` 一样，`FromSqlInterpolated` 只能在查询根上使用。与上述示例一样，该值会转换为 `DbParameter`，且不易受到 SQL 注入攻击。

NOTE

在版本 3.0 之前，`FromSqlRaw` 和 `FromSqlInterpolated` 是名为 `FromSql` 的两个重载。有关详细信息，请参阅[历史版本部分](#)。

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSqlInterpolated($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
    .ToList();
```

还可以构造 `SqlParameter` 并将其作为参数值提供。由于使用了常规 SQL 参数占位符而不是字符串占位符，因此可安全地使用 `FromSqlRaw`：

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser @user", user)
    .ToList();
```

借助 `FromSqlRaw`，可以在 SQL 查询字符串中使用已命名的参数，这在存储的流程具有可选参数时非常有用：

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetMostPopularBlogsForUser @filterByUser=@user", user)
    .ToList();
```

使用 LINQ 编写

可使用 LINQ 运算符在初始的原始 SQL 查询基础上进行组合。EF Core 将其视为子查询，并在数据库中对其进行组合。下面的示例使用原始 SQL 查询，该查询从表值函数 (TVF) 中进行选择。然后，使用 LINQ 进行筛选和排序，从而对其进行组合。

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```

上面的查询生成以下 SQL：

```
SELECT [b].[BlogId], [b].[OwnerId], [b].[Rating], [b].[Url]
FROM (
    SELECT * FROM dbo.SearchBlogs(@p0)
) AS [b]
WHERE [b].[Rating] > 3
ORDER BY [b].[Rating] DESC
```

包含关联数据

`Include` 方法可用于添加相关数据，就像对其他 LINQ 查询那样：

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Include(b => b.Posts)
    .ToList();
```

使用 LINQ 进行组合要求原始 SQL 查询是可组合的，因为 EF Core 会将提供的 SQL 视为子查询。以 `SELECT` 关键字开始的 SQL 查询一般是可组合的。此外，传递的 SQL 不应包含子查询上无效的任何字符或选项，如：

- 结尾分号
- 在 SQL Server 上，结尾处的查询级提示（例如，`OPTION (HASH JOIN)`）
- 在 SQL Server 上，`ORDER BY` 子句中不与 `OFFSET 0` 或 `TOP 100 PERCENT` 配合使用的 `SELECT` 子句

SQL Server 不允许对存储过程调用进行组合，因此任何尝试向此类调用应用其他查询运算符的操作都将导致无效的 SQL。请在 `AsEnumerable` 或 `AsAsyncEnumerable` 方法之后立即使用 `FromSqlRaw` 或 `FromSqlInterpolated` 方法，确保 EF Core 不会尝试对存储过程进行组合。

更改跟踪

使用 `FromSqlRaw` 或 `FromSqlInterpolated` 方法的查询遵循与 EF Core 中所有其他 LINQ 查询完全相同的更改跟踪规则。例如，如果该查询投影实体类型，默认情况下会跟踪结果。

下面的示例使用原始 SQL 查询，该查询从表值函数 (TVF) 中进行选择，然后禁用通过对 `AsNoTracking` 的调用来更改跟踪：

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSqlInterpolated($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .AsNoTracking()
    .ToList();
```

限制

使用原生 SQL 查询时需注意以下几个限制：

- SQL 查询必须返回实体类型的所有属性的数据。
- 结果集中的列名必须与属性映射到的列名匹配。请注意，此行为不同于 EF6。EF6 中忽略了原始 SQL 查询的属性/列映射关系，结果集列名必须与属性名相匹配。
- SQL 查询不能包含关联数据。但是，在许多情况下你可以在查询后面紧跟着使用 `Include` 方法以返回关联数据（请参阅[包含关联数据](#)）。

以前的版本

EF Core 2.2 及更低版本具有两个名为 `FromSql` 的方法重载，它们的行为方式与较新的 `FromSqlRaw` 和 `FromSqlInterpolated` 的相同。因此，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。意外调用错误的重载会导致本该参数化的查询没有参数化。

全局查询筛选器

2020/4/8 • [Edit Online](#)

NOTE

EF Core 2.0 中已引入此功能。

全局查询筛选器是应用于元数据模型(通常为 *OnModelCreating*)中的实体类型的 LINQ 查询谓词(通常传递给 LINQ *Where* 查询运算符的布尔表达式)。此类筛选器自动应用于涉及这些实体类型(包括通过使用 *Include* 或直接导航属性引用等方式间接引用的实体类型)的所有 LINQ 查询。此功能的一些常见应用如下:

- **软删除** - 实体型定义“*IsDeleted*”属性。
- **多租户** - 实体型定义“*TenantId*”属性。

示例

下面的示例显示了如何使用全局查询筛选器在简单的博客模型中实现软删除和多租户查询行为。

TIP

可在 GitHub 上查看此文章的示例。

首先, 定义实体:

```
public class Blog
{
    private string _tenantId;

    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public bool IsDeleted { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

请注意 *Blog* 实体上的 *tenantId* 字段的声明。这会用于将每个 *Blog* 实例与特定租户相关联。还会定义 *文章* 实体类型上的 *IsDeleted* 属性。这会用于跟踪文章实例是否已“软删除”。也就是说, 实例标记为已删除, 而实际上不会删除基础数据。

接下来, 使用 `_API` 在 `_OnModelCreating` 中配置查询筛选器。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().Property<string>("_tenantId").HasColumnName("TenantId");

    // Configure entity filters
    modelBuilder.Entity<Blog>().HasQueryFilter(b => EF.Property<string>(b, "_tenantId") == _tenantId);
    modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
}
```

传递给 `HasQueryFilter` 调用的谓词表达式将立即自动应用于这些类型的任何 LINQ 查询。

TIP

请注意 `DbContext` 实例级别字段的使用：`_tenantId` 用于设置当前租户。模型级筛选器将使用正确上下文实例（即执行查询的实例）中的值。

NOTE

目前不能在同一个实体中定义多个查询筛选器，只会应用最后一个筛选器。但是，可以使用逻辑 `AND` 运算符（C# 中为 `&&`）定义含有多种条件的单个筛选器。

禁用筛选器

可使用 `IgnoreQueryFilters()` 运算符对各个 LINQ 查询禁用筛选器。

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

限制

全局查询筛选器具有以下限制：

- 仅可为继承层次结构中的根实体类型定义筛选器。

查询标记

2020/4/8 • [Edit Online](#)

NOTE

此为 EF Core 2.2 中的新增功能。

此功能有助于将代码中的 LINQ 查询与日志中捕获的已生成 SQL 查询相关联。使用新增的 `TagWith()` 方法对 LINQ 查询进行批注：

```
var nearestFriends =
    (from f in context.Friends.TagWith("This is my spatial query!")
     orderby f.Location.Distance(myLocation) descending
     select f).Take(5).ToList();
```

此 LINQ 查询转换为以下 SQL 语句：

```
-- This is my spatial query!

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

可以对同一个查询多次调用 `TagWith()`。查询标记具有累积性。例如，给定方法如下：

```
IQueryable<Friend> GetNearestFriends(Point myLocation) =>
    from f in context.Friends.TagWith("GetNearestFriends")
    orderby f.Location.Distance(myLocation) descending
    select f;

IQueryable<T> Limit<T>(IQueryable<T> source, int limit) =>
    source.TagWith("Limit").Take(limit);
```

以下查询：

```
var results = Limit(GetNearestFriends(myLocation), 25).ToList();
```

转换为：

```
-- GetNearestFriends

-- Limit

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

也可以使用多线串作为查询标记。例如：

```
var results = Limit(GetNearestFriends(myLocation), 25).TagWith(
    @"This is a multi-line
    string").ToList();
```

生成以下 SQL:

```
-- GetNearestFriends

-- Limit

-- This is a multi-line
-- string

SELECT TOP(@__p_1) [f].[Name], [f].[Location]
FROM [Friends] AS [f]
ORDER BY [f].[Location].STDistance(@__myLocation_0) DESC
```

已知的限制

查询标记不可参数化: EF Core 始终将 LINQ 查询中的查询标记视为, 已生成 SQL 中包含的字符串文本。禁止使用将查询标记用作参数的已编译查询。

查询的工作原理

2020/4/8 • [Edit Online](#)

Entity Framework Core 使用语言集成查询 (LINQ) 来查询数据库中的数据。通过 LINQ 可使用 C#(或你选择的其他 .NET 语言) 基于派生上下文和实体类编写强类型查询。

查询的生命周期

下面是每个查询所经历的过程的高级概述。

1. LINQ 查询由 Entity Framework Core 处理, 用于生成已准备好由数据库提供程序处理的表示形式
 - a. 结果会被缓存, 以便每次执行查询时无需重复进行此处理
2. 结果会传递到数据库提供程序
 - a. 数据库提供程序会识别出查询的哪些部分可以在数据库中求值
 - b. 查询的这些部分会转换为特定数据库的查询语言(例如, 关系数据库的 SQL)
 - c. 查询会发送到数据库并返回结果集(返回的是数据库中的值, 而不是实体实例中的)
3. 对于结果集中的每一项
 - a. 如果这是跟踪查询, EF 会检查数据是否表示上下文实例内更改跟踪器中的现有实体
 - 如果是, 则会返回现有实体
 - 如果不是, 则会创建新实体、设置更改跟踪并返回该新实体
 - b. 如果这是 no-tracking 查询, 将始终创建并返回新实体

执行查询时

调用 LINQ 运算符时, 只会构建查询在内存中的表示形式。只有在使用结果时, 查询才会发送到数据库。

导致查询发送到数据库的最常见操作如下:

- 在 `for` 循环中循环访问结果
- 使用 `ToList`、`ToDictionary`、`Single`、`Count` 等操作或等效的异步重载

WARNING

始终验证用户输入: 虽然 EF Core 通过在查询中使用参数和转义文字来防止 SQL 注入攻击, 但它不会验证输入。根据应用程序的要求, 在将 LINQ 查询中使用的来自不受信任的源的值分配给实体属性或传递给其他 EF Core API 之前, 应执行相应的验证。这包括用于动态构造查询的所有用户输入。即使在使用 LINQ 时, 如果接受用于生成表达式的用户输入, 也会需要确保只能构造预期表达式。

保存数据

2020/4/8 • [Edit Online](#)

每个上下文实例都有一个 `ChangeTracker`，它负责跟踪需要写入数据库的更改。更改实体类的实例时，这些更改会记录在 `ChangeTracker` 中，然后在调用 `SaveChanges` 时被写入数据库。此数据库提供程序负责将更改转换为特定于数据库的操作（例如，关系数据库的 `INSERT`、`UPDATE` 和 `DELETE` 命令）。

基本保存

2020/4/8 • [Edit Online](#)

了解如何使用上下文和实体类添加、修改和删除数据。

TIP

可在 GitHub 上查看此文章的示例。

添加数据

使用 `DbSet.Add` 方法添加实体类的新实例。调用 `SaveChanges` 时，数据将插入到数据库中。

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://example.com" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

TIP

添加、附加和更新方法全部呈现在传递给这些方法的实体的完整关系图上，如[相关数据部分](#)中所述。此外，还可以使用 `EntityEntry.State` 属性仅设置单个实体的状态。例如，`context.Entry(blog).State = EntityState.Modified`。

更新数据

EF 将自动检测对由上下文跟踪的现有实体所做的更改。这包括从数据库加载/查询的实体，以及之前添加并保存到数据库的实体。

只需通过赋值来修改属性，然后调用 `SaveChanges` 即可。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    blog.Url = "http://example.com/blog";
    context.SaveChanges();
}
```

删除数据

使用 `DbSet.Remove` 方法删除实体类的实例。

如果实体已存在于数据库中，则将在“`SaveChanges`”期间删除该实体。如果实体尚未保存到数据库（即跟踪为“已添加”），则在调用 `SaveChanges` 时，该实体会从上下文中移除且不再插入。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    context.Blogs.Remove(blog);
    context.SaveChanges();
}
```

单个 SaveChanges 中的多个操作

可以将多个添加/更新/删除操作合并到对“SaveChanges”的单个调用。

NOTE

对于大多数数据库提供程序，“SaveChanges”是事务性的。这意味着所有操作将一起成功或一起失败，绝不会部分的应用这些操作。

```
using (var context = new BloggingContext())
{
    // seeding database
    context.Blogs.Add(new Blog { Url = "http://example.com/blog" });
    context.Blogs.Add(new Blog { Url = "http://example.com/another_blog" });
    context.SaveChanges();
}

using (var context = new BloggingContext())
{
    // add
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://example.com/blog_two" });

    // update
    var firstBlog = context.Blogs.First();
    firstBlog.Url = "";

    // remove
    var lastBlog = context.Blogs.Last();
    context.Blogs.Remove(lastBlog);

    context.SaveChanges();
}
```

保存相关数据

2020/4/8 • [Edit Online](#)

除了独立实体以外，还可以使用模型中定义的关系。

TIP

可在 GitHub 上查看此文章的[示例](#)。

添加新实体的关系图

如果创建多个新的相关实体，则将其中一个添加到上下文时也会添加其他实体。

在下面的示例中，博客和三篇相关文章将全部被插入数据库中。由于可通过 `Blog.Posts` 导航属性访问这些文章，因此可发现并添加它们。

```
using (var context = new BloggingContext())
{
    var blog = new Blog
    {
        Url = "http://blogs.msdn.com/dotnet",
        Posts = new List<Post>
        {
            new Post { Title = "Intro to C#" },
            new Post { Title = "Intro to VB.NET" },
            new Post { Title = "Intro to F#" }
        }
    };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

TIP

使用 `EntityEntry.State` 属性仅设置单个实体的状态。例如，`context.Entry(blog).State = EntityState.Modified`。

添加相关实体

如果从已由上下文跟踪的实体的导航属性中引用新实体，则将发现该实体并将其插入到数据库中。

在下面的示例中，插入 `post` 实体，因为该实体会添加到已从数据库中提取的 `Posts` 实体的 `blog` 属性。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = new Post { Title = "Intro to EF Core" };

    blog.Posts.Add(post);
    context.SaveChanges();
}
```

更改关系

如果更改实体的导航属性，则将对数据库中的外键列进行相应的更改。

在下面的示例中，`post` 实体更新为属于新的 `blog` 实体，因为其 `Blog` 导航属性设置为指向 `blog`。请注意，`blog` 也会插入到数据库中，因为它是已由上下文跟踪的实体 (`post`) 的导航属性引用的新实体。

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Url = "http://blogs.msdn.com/visualstudio" };
    var post = context.Posts.First();

    post.Blog = blog;
    context.SaveChanges();
}
```

删除关系

可以通过将引用导航设置为 `null` 或从集合导航中删除相关实体来删除关系。

根据关系中配置的级联删除行为，删除关系可能会对依赖实体产生副作用。

默认情况下，对于必选关系，将配置级联删除行为，并将从数据库中删除子实体/依赖实体。对于可选关系，默认情况下不会配置级联删除，但会将外键属性设置为 `null`。

要了解有关如何配置关系必要性的信息，请参阅[必选关系和可选关系](#)。

有关级联删除行为的工作原理、如何显式配置这些行为以及如何按照约定选择这些行为的详细信息，请参阅[级联删除](#)。

在下面的示例中，对 `Blog` 和 `Post` 之间的关系配置了级联删除，因此将从数据库中删除 `post` 实体。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Include(b => b.Posts).First();
    var post = blog.Posts.First();

    blog.Posts.Remove(post);
    context.SaveChanges();
}
```

级联删除

2020/4/8 • [Edit Online](#)

级联删除通常在数据库术语中用来描述一种允许在删除某行时自动触发删除相关行的特性。EF Core 删除行为还介绍了一个密切相关的概念，即子实体与父实体的关系已断开时自动删除该子实体，这通常称为“删除孤立项”。

EF Core 实现多种不同的删除行为，并允许配置各个关系的删除行为。EF Core 还实现基于[关系的需求](#)为每个关系自动配置有用，默认删除行为的约定。

删除行为

删除行为在 `DeleteBehavior` 枚举器类型中定义，并且可以传递到 `OnDelete` Fluent API 来控制是主体/父实体的删除还是依赖实体/子实体关系的断开会对依赖实体/子实体产生副作用。

删除主体/父实体或断开与子实体的关系时有三个 EF 可执行的操作：

- 可以删除子项/依赖项
- 子项的外键值可以设置为 null
- 子项保持不变

NOTE

仅当使用 EF Core 删除主体且将依赖实体加载到内存中（即对于跟踪的依赖项）时才应用 EF Core 模型中配置的删除行为。需要在数据库中设置相应的级联行为以确保未由上下文跟踪的数据已应用必要的操作。如果使用 EF Core 创建数据库，将为你设置此级联行为。

对于上述第二个操作，如果外键不可以为 null，则将外键值设置为 null 是无效的。（不可为 null 的外键相当于必需关系。）在这些情况下，EF Core 会跟踪外键属性是否已被标记为 null，直到调用 `SaveChanges`，此时会引发异常，因为无法将更改永久保存到数据库中。这类似于从数据库中获取约束冲突。

有四个删除行为，如下表中列出。

可选关系

对于可选关系（可以为 null 的外键），可以保存 null 外键值，从而产生以下影响：

关系	OnDelete / OnUpdate	OnDelete / OnUpdate
Cascade	删除实体	删除实体
ClientSetNull (默认)	外键属性设置为 null	None
SetNull	外键属性设置为 null	外键属性设置为 null
Restrict	None	None

必选关系

对于必选关系（不可为 null 的外键），不可以保存 null 外键值，从而产生以下影响：

外键	主键/外键	外键/主键
Cascade(默认)	删除实体	删除实体
ClientSetNull	SaveChanges 引发异常	None
SetNull	引发 SaveChanges	SaveChanges 引发异常
Restrict	None	None

在上表中，“无”可能会造成约束冲突。例如，如果已删除主体/子实体，但不执行任何操作来更改依赖项/子项的外键，则由于发生外键约束冲突，数据库将可能会引发 SaveChanges。

高级别：

- 如果实体在没有父项时不能存在，且希望 EF 负责自动删除子项，则使用“Cascade”。
 - 在没有父项时不能存在的实体通常使用必选关系，其中“Cascade”是默认值。
- 如果实体可能有或可能没有父项，且希望 EF 负责为你将外键变为 null，则使用“ClientSetNull”
 - 在没有父项时可以存在的实体通常使用可选关系，其中“ClientSetNull”是默认值。
 - 如果希望数据库即使在未加载子实体时也尝试将 null 值传播到子外键，则使用“SetNull”。但是，请注意，数据库必须支持此操作，并且如此配置数据库可能会导致其他限制，实际上这通常会使此选项不适用。这就是 SetNull 不是默认值的原因。
- 如果不希望 EF Core 始终自动删除实体或自动将外键变为 null，则使用“Restrict”。请注意，这要求使用代码手动同步子实体及其外键值，否则将引发约束异常。

NOTE

在 EF Core(与 EF6 不同)中，不会立即产生级联影响，而是仅在调用 SaveChanges 时产生。

NOTE

EF Core 2.0 ■：在之前的版本中，“Restrict”将导致跟踪的依赖实体中的可选外键属性设置为 null，并且是可选关系的默认删除行为。在 EF Core 2.0 中，引入了“ClientSetNull”以表示该行为，并且它会成为可选关系的默认值。“Restrict”的行为已调整为永远不会对依赖实体产生副作用。

实体删除示例

以下代码是可下载并运行的[示例](#)的一部分。此示例显示了，当删除父实体时，可选关系和必选关系的每个删除行为会发生的情况。

```

var blog = context.Blogs.Include(b => b.Posts).First();
var posts = blog.Posts.ToList();

DumpEntities(" After loading entities:", context, blog, posts);

context.Remove(blog);

DumpEntities($" After deleting blog '{blog.BlogId}':", context, blog, posts);

try
{
    Console.WriteLine();
    Console.WriteLine(" Saving changes:");

    context.SaveChanges();

    DumpSql();

    DumpEntities(" After SaveChanges:", context, blog, posts);
}
catch (Exception e)
{
    DumpSql();

    Console.WriteLine();
    Console.WriteLine($" SaveChanges threw {e.GetType().Name}: {(e is DbUpdateException ? e.InnerException.Message : e.Message)}");
}

```

我们来看一看每个变化以了解所发生的情况。

具有必选或可选关系的 DeleteBehavior.Cascade

```

After loading entities:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

After deleting blog '1':
Blog '1' is in state Deleted with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

Saving changes:
DELETE FROM [Posts] WHERE [PostId] = 1
DELETE FROM [Posts] WHERE [PostId] = 2
DELETE FROM [Blogs] WHERE [BlogId] = 1

After SaveChanges:
Blog '1' is in state Detached with 2 posts referenced.
Post '1' is in state Detached with FK '1' and no reference to a blog.
Post '2' is in state Detached with FK '1' and no reference to a blog.

```

- 博客标记为已删除
- 文章最初保持不变，因为在调用 SaveChanges 之前不会发生级联
- SaveChanges 发送对依赖项/子项(文章)和主体/父项(博客)的删除
- 保存后，所有实体都会分离，因为它们现在已从数据库中删除

具有必选关系的 DeleteBehavior.ClientSetNull 或 DeleteBehavior.SetNull

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After deleting blog '1':  
Blog '1' is in state Deleted with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
Saving changes:  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1
```

```
SaveChanges threw DbUpdateException: Cannot insert the value NULL into column 'BlogId', table  
'EFSaving.CascadeDelete.dbo.Posts'; column does not allow nulls. UPDATE fails. The statement has been  
terminated.
```

- 博客标记为已删除
- 文章最初保持不变，因为在调用 SaveChanges 之前不会发生级联
- SaveChanges 尝试将文章外键设置为 null，但会失败，因为外键不可以为 null

具有可选关系的 DeleteBehavior.ClientSetNull 或 DeleteBehavior.SetNull

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After deleting blog '1':  
Blog '1' is in state Deleted with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
Saving changes:  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 2  
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

```
After SaveChanges:  
Blog '1' is in state Detached with 2 posts referenced.  
Post '1' is in state Unchanged with FK 'null' and no reference to a blog.  
Post '2' is in state Unchanged with FK 'null' and no reference to a blog.
```

- 博客标记为已删除
- 文章最初保持不变，因为在调用 SaveChanges 之前不会发生级联
- SaveChanges 尝试在删除主体/父项(博客)之前将依赖项/子项(文章)的外键设置为 null
- 保存后，将删除主体/父项(博客)，但仍会跟踪依赖项/子项(文章)
- 跟踪的依赖项/子项(文章)现在具有 null 外键值，并且对删除的主体/父项(博客)的引用已删除

具有必选或可选关系的 DeleteBehavior.Restrict

```

After loading entities:
Blog '1' is in state Unchanged with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

After deleting blog '1':
Blog '1' is in state Deleted with 2 posts referenced.
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.

Saving changes:
SaveChanges threw InvalidOperationException: The association between entity types 'Blog' and 'Post' has
been severed but the foreign key for this relationship cannot be set to null. If the dependent entity should
be deleted, then setup the relationship to use cascade deletes.

```

- 博客标记为已删除
- 文章最初保持不变，因为在调用 SaveChanges 之前不会发生级联
- 由于 `Restrict` 指示 EF 不要自动将外键设置为 null，因此它保持为非 null，而 SaveChanges 将引发异常但不进行保存

删除孤立项示例

以下代码是可下载并运行的[示例](#)的一部分。此示例显示了，当断开父项/主体及其子项/依赖项之间的关系时，可选关系和必选关系的每个删除行为会发生的情况。在此示例中，通过从主体/父项(博客)上的集合导航属性中删除依赖项/子项(文章)来断开关系。但是，如果将从依赖项/子项到主体/父项的引用变为 null，则行为相同。

```

var blog = context.Blogs.Include(b => b.Posts).First();
var posts = blog.Posts.ToList();

DumpEntities(" After loading entities:", context, blog, posts);

blog.Posts.Clear();

DumpEntities(" After making posts orphans:", context, blog, posts);

try
{
    Console.WriteLine();
    Console.WriteLine(" Saving changes:");

    context.SaveChanges();

    DumpSql();

    DumpEntities(" After SaveChanges:", context, blog, posts);
}
catch (Exception e)
{
    DumpSql();

    Console.WriteLine();
    Console.WriteLine($" SaveChanges threw {e.GetType().Name}: {(e is DbUpdateException ? e.InnerException.Message : e.Message)}");
}

```

我们来看一看每个变化以了解所发生的情况。

具有必选或可选关系的 DeleteBehavior.Cascade

After loading entities:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

After making posts orphans:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK '1' and no reference to a blog.  
Post '2' is in state Modified with FK '1' and no reference to a blog.
```

Saving changes:

```
DELETE FROM [Posts] WHERE [PostId] = 1  
DELETE FROM [Posts] WHERE [PostId] = 2
```

After SaveChanges:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Detached with FK '1' and no reference to a blog.  
Post '2' is in state Detached with FK '1' and no reference to a blog.
```

- 文章标记为已修改, 因为断开关系导致外键标记为 null
 - 如果外键不可以为 null, 则即使实际值标记为 null 也不会更改
- SaveChanges 发送对依赖项/子项(文章)的删除
- 保存后, 依赖项/子项(文章)会分离, 因为它们现在已从数据库中删除

具有必选关系的 DeleteBehavior.ClientSetNull 或 DeleteBehavior.SetNull

After loading entities:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

After making posts orphans:

```
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK 'null' and no reference to a blog.  
Post '2' is in state Modified with FK 'null' and no reference to a blog.
```

Saving changes:

```
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1
```

```
SaveChanges threw DbUpdateException: Cannot insert the value NULL into column 'BlogId', table  
'EFSaving.CascadeDelete.dbo.Posts'; column does not allow nulls. UPDATE fails. The statement has been  
terminated.
```

- 文章标记为已修改, 因为断开关系导致外键标记为 null
 - 如果外键不可以为 null, 则即使实际值标记为 null 也不会更改
- SaveChanges 尝试将文章外键设置为 null, 但会失败, 因为外键不可以为 null

具有可选关系的 DeleteBehavior.ClientSetNull 或 DeleteBehavior.SetNull

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After making posts orphans:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK 'null' and no reference to a blog.  
Post '2' is in state Modified with FK 'null' and no reference to a blog.
```

```
Saving changes:  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 1  
UPDATE [Posts] SET [BlogId] = NULL WHERE [PostId] = 2
```

```
After SaveChanges:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK 'null' and no reference to a blog.  
Post '2' is in state Unchanged with FK 'null' and no reference to a blog.
```

- 文章标记为已修改, 因为断开关系导致外键标记为 null
 - 如果外键不可以为 null, 则即使实际值标记为 null 也不会更改
- SaveChanges 将依赖项/子项(文章)的外键设置为 null
- 保存后, 依赖项/子项(文章)现在具有 null 外键值, 并且对删除的主体/父项(博客)的引用已删除

具有必选或可选关系的 DeleteBehavior.Restrict

```
After loading entities:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Unchanged with FK '1' and reference to blog '1'.  
Post '2' is in state Unchanged with FK '1' and reference to blog '1'.
```

```
After making posts orphans:  
Blog '1' is in state Unchanged with 2 posts referenced.  
Post '1' is in state Modified with FK '1' and no reference to a blog.  
Post '2' is in state Modified with FK '1' and no reference to a blog.
```

```
Saving changes:  
SaveChanges threw InvalidOperationException: The association between entity types 'Blog' and 'Post' has  
been severed but the foreign key for this relationship cannot be set to null. If the dependent entity should  
be deleted, then setup the relationship to use cascade deletes.
```

- 文章标记为已修改, 因为断开关系导致外键标记为 null
 - 如果外键不可以为 null, 则即使实际值标记为 null 也不会更改
- 由于 Restrict 指告知 EF 不要自动将外键设置为 null, 因此它保持为非 null, 而 SaveChanges 将引发异常但不进行保存

级联到未跟踪的实体

调用“SaveChanges”时, 级联删除规则将应用于由上下文跟踪的所有实体。这是上述所有示例的情况, 即生成用于删除主体/父项(博客)和所有依赖项/子项(文章)的 SQL 的原因:

```
DELETE FROM [Posts] WHERE [PostId] = 1  
DELETE FROM [Posts] WHERE [PostId] = 2  
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

如果仅加载主体(例如, 当为不含 `Include(b => b.Posts)` 的博客创建查询以包含文章时), 则 SaveChanges 只会生成用于删除主体/父项的 SQL:

```
DELETE FROM [Blogs] WHERE [BlogId] = 1
```

如果数据库已配置相应的级联行为，则只会删除依赖项/子项(文章)。如果使用 EF 创建数据库，则会为你设置此级联行为。

处理并发冲突

2020/4/8 • [Edit Online](#)

NOTE

此页面记录了并发在 EF Core 中的工作原理以及如何处理应用程序中的并发冲突。有关如何配置模型中的并发令牌的详细信息，请参阅[并发令牌](#)。

TIP

可在 GitHub 上查看此文章的示例。

数据库并发指多个进程或用户同时访问或更改数据库中的相同数据的情况。并发控制指的是用于在发生并发更改时确保数据一致性的特定机制。

EF Core 实现了乐观并发控制，这意味着它将允许多个进程或用户独立进行更改，而不会产生同步或锁定的开销。在理想情况下，这些更改将不会相互干扰，因此都能够成功。在最坏的情况下，两个或更多进程将尝试进行冲突更改，而其中只有一个进程会成功。

并发控制在 EF Core 中的工作原理

配置为并发令牌的属性用于实现乐观并发控制：每当在 `SaveChanges` 期间执行更新或删除操作时，会将数据库上的并发令牌值与通过 EF Core 读取的原始值进行比较。

- 如果这些值匹配，则可以完成该操作。
- 如果这些值不匹配，EF Core 会假设另一个用户已执行冲突操作，并中止当前事务。

另一个用户已执行与当前操作冲突的操作的情况被称为并发冲突。

数据库提供程序负责实现并发令牌值的比较。

在关系数据库上，EF Core 会对任何 `WHERE` 或 `UPDATE` 语句的 `DELETE` 子句中的并发令牌值进行检查。执行这些语句后，EF Core 会读取受影响的行数。

如果未影响任何行，将检测到并发冲突，并且 EF Core 会引发 `DbUpdateConcurrencyException`。

例如，我们可能希望将 `LastName` 的 `Person` 配置为并发令牌。这样，对 `Person` 的任何更新操作都将在 `WHERE` 子句中包括并发检查：

```
UPDATE [Person] SET [FirstName] = @p1  
WHERE [PersonId] = @p0 AND [LastName] = @p2;
```

解决并发冲突

继续前面的示例，如果一个用户尝试保存对 `Person` 所做的某些更改，但另一个用户已更改 `LastName`，则将引发异常。

此时，应用程序可能只需通知用户由于发生冲突更改而导致更新未成功，然后继续操作。但可能需要提示用户确保此记录仍表示同一实际用户并重试该操作。

此过程是解决并发冲突的一个示例。

解决并发冲突涉及将当前 `DbContext` 中挂起的更改与数据库中的值进行合并。要合并的值因应用程序而异并可由用户输入指示。

有三组值可用于帮助解决并发冲突：

- “当前值”是应用程序尝试写入数据库的值。
- “原始值”是在进行任何编辑之前最初从数据库中检索的值。
- “数据库值”是当前存储在数据库中的值。

处理并发冲突的常规方法是：

1. 在 `DbUpdateConcurrencyException` 期间捕获 `SaveChanges`。
2. 使用 `DbUpdateConcurrencyException.Entries` 为受影响的实体准备一组新更改。
3. 刷新并发令牌的原始值以反映数据库中的当前值。
4. 重试该过程，直到不发生任何冲突。

在下面的示例中，将 `Person.FirstName` 和 `Person.LastName` 设置为并发令牌。在包括应用程序特定逻辑以选择要保存的值的位置处有一条 `// TODO:` 注释。

```

using (var context = new PersonContext())
{
    // Fetch a person from database and change phone number
    var person = context.People.Single(p => p.PersonId == 1);
    person.PhoneNumber = "555-555-5555";

    // Change the person's name in the database to simulate a concurrency conflict
    context.Database.ExecuteSqlRaw(
        "UPDATE dbo.People SET FirstName = 'Jane' WHERE PersonId = 1");

    var saved = false;
    while (!saved)
    {
        try
        {
            // Attempt to save changes to the database
            context.SaveChanges();
            saved = true;
        }
        catch (DbUpdateConcurrencyException ex)
        {
            foreach (var entry in ex.Entries)
            {
                if (entry.Entity is Person)
                {
                    var proposedValues = entry.CurrentValues;
                    var databaseValues = entry.GetDatabaseValues();

                    foreach (var property in proposedValues.Properties)
                    {
                        var proposedValue = proposedValues[property];
                        var databaseValue = databaseValues[property];

                        // TODO: decide which value should be written to database
                        // proposedValues[property] = <value to be saved>;
                    }

                    // Refresh original values to bypass next concurrency check
                    entry.OriginalValues.SetValues(databaseValues);
                }
                else
                {
                    throw new NotSupportedException(
                        "Don't know how to handle concurrency conflicts for "
                        + entry.Metadata.Name);
                }
            }
        }
    }
}

```

使用事务

2020/4/8 • [Edit Online](#)

事务允许以原子方式处理多个数据库操作。如果已提交事务，则所有操作都会成功应用到数据库。如果已回滚事务，则所有操作都不会应用到数据库。

TIP

可在 GitHub 上查看此文章的示例。

默认事务行为

默认情况下，如果数据库提供程序支持事务，则会在单次调用 `SaveChanges()` 时将所有更改都将应用到事务中。如果其中有任何更改失败，则会回滚事务且所有更改都不会应用到数据库。这意味着，`SaveChanges()` 可保证要么完全成功，要么在出现错误时不修改数据库。

对于大多数应用程序，此默认行为已足够。除非应用程序确有需求，否则不应手动控制事务。

控制事务

可以使用 `DbContext.Database` API 开始、提交和回滚事务。以下示例显示了在单个事务中执行的两个 `SaveChanges()` 操作以及一个 LINQ 查询。

并非所有数据库提供程序都支持事务。调用事务 API 时，某些提供程序可能会引发异常或不执行任何操作。

```
using (var context = new BloggingContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context.SaveChanges();

            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/visualstudio" });
            context.SaveChanges();

            var blogs = context.Blogs
                .OrderBy(b => b.Url)
                .ToList();

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

跨上下文事务（仅限关系数据库）

您还可以跨多个上下文实例共享一个事务。此功能仅在使用关系数据库提供程序时才可用，因为它需要使用特定于关系数据库的 `DbTransaction` 和 `DbConnection`。

若要共享事务，上下文必须共享 `DbConnection` 和 `DbTransaction`。

允许在外部提供连接

共享 `DbConnection` 需要在构造上下文时向其中传入连接的功能。

允许在外部提供 `DbConnection` 的最简单方式是，停止使用 `DbContext.OnConfiguring` 方法来配置上下文并在外部创建 `DbContextOptions`，然后将其传递到上下文构造函数。

TIP

`DbContextOptionsBuilder` 是在 `DbContext.OnConfiguring` 中用于配置上下文的 API，现在即将在外部使用它来创建 `DbContextOptions`。

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

替代方法是继续使用 `DbContext.OnConfiguring`，但接受已保存并随后在 `DbConnection` 中使用的 `DbContext.OnConfiguring`。

```
public class BloggingContext : DbContext
{
    private DbConnection _connection;

    public BloggingContext(DbConnection connection)
    {
        _connection = connection;
    }

    public DbSet<Blog> Blogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(_connection);
    }
}
```

共享连接和事务

现在可以创建共享同一连接的多个上下文实例。然后使用 `DbContext.Database.UseTransaction(DbTransaction)` API 在同一事务中登记两个上下文。

```
var options = new DbContextOptionsBuilder<BloggingContext>()
    .UseSqlServer(new SqlConnection(connectionString))
    .Options;

using (var context1 = new BloggingContext(options))
{
    using (var transaction = context1.Database.BeginTransaction())
    {
        try
        {
            context1.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context1.SaveChanges();

            using (var context2 = new BloggingContext(options))
            {
                context2.Database.UseTransaction(transaction.GetDbTransaction());

                var blogs = context2.Blogs
                    .OrderBy(b => b.Url)
                    .ToList();
            }

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

使用外部 DbTransactions(仅限关系数据库)

如果使用多个数据访问技术来访问关系数据库，则可能希望在这些不同技术所执行的操作之间共享事务。

以下示例显示了如何在同一事务中执行 ADO.NET SqlClient 操作和 Entity Framework Core 操作。

```
using (var connection = new SqlConnection(connectionString))
{
    connection.Open();

    using (var transaction = connection.BeginTransaction())
    {
        try
        {
            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.Transaction = transaction;
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseSqlServer(connection)
                .Options;

            using (var context = new BloggingContext(options))
            {
                context.Database.UseTransaction(transaction);
                context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context.SaveChanges();
            }

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            transaction.Commit();
        }
        catch (System.Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

使用 System.Transactions

NOTE

此功能是 EF Core 2.1 中的新增功能。

如果需要跨较大作用域进行协调，则可以使用环境事务。

```
using (var scope = new TransactionScope(
    TransactionScopeOption.Required,
    new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted }))
{
    using (var connection = new SqlConnection(connectionString))
    {
        connection.Open();

        try
        {
            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            var options = new DbContextOptionsBuilder<BloggingContext>()
                .UseSqlServer(connection)
                .Options;

            using (var context = new BloggingContext(options))
            {
                context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
                context.SaveChanges();
            }

            // Commit transaction if all commands succeed, transaction will auto-rollback
            // when disposed if either commands fails
            scope.Complete();
        }
        catch (System.Exception)
        {
            // TODO: Handle failure
        }
    }
}
```

还可以在显式事务中登记。

```

using (var transaction = new CommittableTransaction(
    new TransactionOptions { IsolationLevel = IsolationLevel.ReadCommitted }))
{
    var connection = new SqlConnection(connectionString);

    try
    {
        var options = new DbContextOptionsBuilder<BloggingContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggingContext(options))
        {
            context.Database.OpenConnection();
            context.Database.EnlistTransaction(transaction);

            // Run raw ADO.NET command in the transaction
            var command = connection.CreateCommand();
            command.CommandText = "DELETE FROM dbo.Blogs";
            command.ExecuteNonQuery();

            // Run an EF Core command in the transaction
            context.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/dotnet" });
            context.SaveChanges();
            context.Database.CloseConnection();
        }

        // Commit transaction if all commands succeed, transaction will auto-rollback
        // when disposed if either commands fails
        transaction.Commit();
    }
    catch (System.Exception)
    {
        // TODO: Handle failure
    }
}

```

System.Transactions 的限制

1. EF Core 依赖数据库提供程序以实现对 System.Transactions 的支持。虽然支持在 .NET Framework 的 ADO.NET 提供程序之间十分常见，但最近才将 API 添加到 .NET Core，因此支持并未得到广泛应用。如果提供程序未实现对 System.Transactions 的支持，则可能会完全忽略对这些 API 的调用。SqlClient for .NET Core 从 2.1 及以上版本开始支持 System.Transactions。如果你尝试使用此功能，SqlClient for .NET Core 2.0 会抛出异常。

IMPORTANT

建议你测试在依赖提供程序以管理事务之前 API 与该提供程序的行为是否正确。如果不正确，则建议你与数据库提供程序的维护人员联系。

2. 自版本 2.1 起，.NET Core 中的 System.Transactions 实现不包括对分布式事务的支持，因此不能使用 `TransactionScope` 或 `CommittableTransaction` 来跨多个资源管理器协调事务。

异步保存

2020/4/8 • [Edit Online](#)

当所做的更改被写入数据库时，异步保存可避免阻塞线程。这有助于避免冻结富客户端应用程序的 UI。异步操作还可以增加 Web 应用程序的吞吐量，可以在数据库操作完成时释放线程去处理其他请求。有关详细信息，请参阅[使用 C# 异步编程](#)。

WARNING

EF Core 不支持在同一上下文实例上运行多个并行操作。应始终等待操作完成，然后再开始下一个操作。这通常是通过在每个异步操作上使用 `await` 关键字完成的。

Entity Framework Core 提供了 `DbContext.SaveChangesAsync()` 作为 `DbContext.SaveChanges()` 的异步替代方法。

```
public static async Task AddBlogAsync(string url)
{
    using (var context = new BloggingContext())
    {
        var blog = new Blog { Url = url };
        context.Blogs.Add(blog);
        await context.SaveChangesAsync();
    }
}
```

断开连接的实体

2020/4/8 • [Edit Online](#)

DbContext 实例将自动跟踪从数据库返回的实体。调用 SaveChanges 时，系统将检测对这些实体所做的更改并根据需要更新数据库。有关详细信息，请参阅[基本保存和相关数据](#)。

但是，有时会使用一个上下文实例查询实体，然后使用其他实例对其进行保存。这通常在“断开连接”的情况下发生，例如 Web 应用程序，此情况下实体被查询、发送到客户端、被修改、在请求中发送回服务器，然后进行保存。在这种情况下，第二个上下文实例需要知道实体是新实体（应插入）还是现有实体（应更新）。

TIP

可在 GitHub 上查看此文章的示例。

TIP

EF Core 只能跟踪具有给定主键值的任何实体的一个实例。为避免这种情况成为一个问题，最好为每个工作单元使用短期上下文，使上下文一开始为空、向其附加实体、保存这些实体，然后释放并放弃该上下文。

标识新实体

客户端标识新实体

油客户端通知服务器实体是新实体还是现有实体，这是最简单的情况。例如，通常插入新实体的请求与更新现有实体的请求不同。

本节的其余部分介绍了需要以其他某种方式确定是插入还是更新的情况。

使用自动生成的键

自动生成的键的值通常可用于确定是否需要插入或更新实体。如果尚未设置密钥（即其值仍为 CLR 默认值 null、零等），则实体必须为新实体且需要插入。另一方面，如果已设置键值，则之前一定已保存该实体，且现在需要更新。换而言之，如果键具有值，则已查询实体并已将其发送到客户端，而现在返回进行更新。

实体类型已知时，可以轻松检查未设置的键：

```
public static bool IsItNew(Blog blog)
=> blog.BlogId == 0;
```

但是，EF 还有一种适用于任何实体类型和键类型的内置方法可用于执行此操作：

```
public static bool IsItNew(DbContext context, object entity)
=> !context.Entry(entity).IsKeySet;
```

TIP

即使实体处于“Added”状态，只要实体由上下文跟踪，就会设置键。这有助于遍历实体图并决定如何处理每个实体（例如在使用 TrackGraph API 时）。键值只能以此处显示的方式使用，然后才能执行任何调用以跟踪实体。

使用其他键

未自动生成键值时，需要使用其他某种机制来确定新实体。有以下两种常规方法：

- 查询实体
- 从客户端传递标志

若要查询实体，只需使用 Find 方法：

```
public static bool IsItNew(BloggingContext context, Blog blog)
    => context.Blogs.Find(blog.BlogId) == null;
```

无法显示用于从客户端传递标志的完整代码，这超出了本文档的范围。在 Web 应用中，这通常意味着对不同操作发出不同请求，或者在请求中传递某些状态，然后在控制器中进行提取。

保存单个实体

如果知道是需要插入还是需要更新，则可以相应地使用 Add 或 Update：

```
public static void Insert(DbContext context, object entity)
{
    context.Add(entity);
    context.SaveChanges();
}

public static void Update(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

但是，如果实体使用自动生成的键值，则 Update 方法可以用于以下两种情况：

```
public static void InsertOrUpdate(DbContext context, object entity)
{
    context.Update(entity);
    context.SaveChanges();
}
```

Update 方法通常将实体标记为更新，而不是插入。但是，如果实体具有自动生成的键且未设置任何键值，则实体会自动标记为插入。

TIP

EF Core 2.0 中已引入此行为。对于早期版本，始终需要显式选择 Add 或 Update。

如果实体不使用自动生成的键，则应用程序必须确定是应插入实体还是应更新实体：例如：

```
public static void InsertOrUpdate(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs.Find(blog.BlogId);
    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
    }

    context.SaveChanges();
}
```

步骤如下：

- 如果 Find 返回 null，则数据库尚未包含具有此 ID 的博客，因此对 Add 的调用会将其标记为插入。
- 如果 Find 返回实体，则该实体存在于数据库中，且上下文现在正在跟踪现有实体
 - 然后，使用 SetValues 将此实体上所有属性的值设置为来自客户端的值。
 - SetValues 调用将根据需要标记要更新的实体。

TIP

SetValues 仅将与跟踪实体中的属性具有不同值的属性标记为“已修改”。这意味着当发送更新时，这意味着当发送更新时，只会更新实际发生更改的列。（如果未发生更改，则根本不会发送任何更新。）

使用图形

标识解析

如上所述，EF Core 只能跟踪具有给定主键值的任何实体的一个实例。使用图形时，理想情况下应创建图形，以便保留此固定对象，且上下文应仅用于一个工作单元。如果图形中包含重复项，则必须先处理该图形，然后再将其发送到 EF 以合并多个实例。如果实例具有冲突值和关系，则以上操作可能并不轻松，因此应尽快在应用程序管道中完成重复项合并以避免冲突解决。

所有新实体/所有现有实体

插入或更新博客及其相关文章的集合是使用图形的一个示例。如果应插入图形中的所有实体，或应更新所有这些实体，则该过程与上述单个实体的过程相同。例如，博客和文章的图形创建如下：

```
var blog = new Blog
{
    Url = "http://sample.com",
    Posts = new List<Post>
    {
        new Post {Title = "Post 1"},
        new Post {Title = "Post 2"},
    }
};
```

可按如下方式插入：

```
public static void InsertGraph(DbContext context, object rootEntity)
{
    context.Add(rootEntity);
    context.SaveChanges();
}
```

对 Add 的调用会将博客和所有文章标记为插入。

同样，如果图形中的所有实体都需要更新，则可以使用 Update：

```
public static void UpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

博客及其所有文章将标记为更新。

新实体和现有实体的组合

即使图形包含需要插入的实体和需要更新的实体的组合，使用自动生成的键，Update 也可以再次用于插入和更新：

```
public static void InsertOrUpdateGraph(DbContext context, object rootEntity)
{
    context.Update(rootEntity);
    context.SaveChanges();
}
```

如果图形、博客或文章中的任何实体未设置键值，Update 会将其标记为插入，而其他所有实体会标记为更新。

如前所述，不使用自动生成的键时，可以使用查询并进行一些处理：

```
public static void InsertOrUpdateGraph(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs
        .Include(b => b.Posts)
        .FirstOrDefault(b => b.BlogId == blog.BlogId);

    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
        foreach (var post in blog.Posts)
        {
            var existingPost = existingBlog.Posts
                .FirstOrDefault(p => p.PostId == post.PostId);

            if (existingPost == null)
            {
                existingBlog.Posts.Add(post);
            }
            else
            {
                context.Entry(existingPost).CurrentValues.SetValues(post);
            }
        }
    }

    context.SaveChanges();
}
```

处理删除

由于实体不存在通常表示应删除该实体，因此删除可能很难处理。解决此问题的一种方法是使用“软删除”，以便将该实体标记为已删除，而不是实际删除。然后，删除将变得与更新相同。可以使用[查询筛选器](#)实现软删除。

对于真删除，常见模式是使用查询模式的扩展来执行本质上为图形差异的操作。例如：

```

public static void InsertUpdateOrDeleteGraph(BloggingContext context, Blog blog)
{
    var existingBlog = context.Blogs
        .Include(b => b.Posts)
        .FirstOrDefault(b => b.BlogId == blog.BlogId);

    if (existingBlog == null)
    {
        context.Add(blog);
    }
    else
    {
        context.Entry(existingBlog).CurrentValues.SetValues(blog);
        foreach (var post in blog.Posts)
        {
            var existingPost = existingBlog.Posts
                .FirstOrDefault(p => p.PostId == post.PostId);

            if (existingPost == null)
            {
                existingBlog.Posts.Add(post);
            }
            else
            {
                context.Entry(existingPost).CurrentValues.SetValues(post);
            }
        }

        foreach (var post in existingBlog.Posts)
        {
            if (!blog.Posts.Any(p => p.PostId == post.PostId))
            {
                context.Remove(post);
            }
        }
    }

    context.SaveChanges();
}

```

TrackGraph

在内部, Add、Attach 和 Update 使用图形遍历, 为每个实体就是否应将其标记为 Added(若要插入)、Modified(若要更新)、Unchanged(不执行任何操作)或 Deleted(若要删除)作出决定。此机制是通过 TrackGraph API 公开的。例如, 假设当客户端发送回实体图形时, 会在每个实体上设置某些标志, 指示应如何处理实体。然后, TrackGraph 可用于处理此标志:

```
public static void SaveAnnotatedGraph(DbContext context, object rootEntity)
{
    context.ChangeTracker.TrackGraph(
        rootEntity,
        n =>
    {
        var entity = (EntityBase)n.Entry.Entity;
        n.Entry.State = entity.IsNew
            ? EntityState.Added
            : entity.IsChanged
                ? EntityState.Modified
                : entity.IsDeleted
                    ? EntityState.Deleted
                    : EntityState.Unchanged;
    });
    context.SaveChanges();
}
```

为了简化示例，标志仅作为实体的一部分显示。通常，标志是 DTO 或包含在请求中的其他某个状态的一部分。

设置已生成属性的显式值

2020/4/8 • [Edit Online](#)

生成的属性是在添加和/或更新实体时由 EF 或数据库生成其值的属性。有关详细信息，请参阅[生成的属性](#)。

在某些情况下，可能希望为已生成属性设置显式值，而不是生成一个显式值。

TIP

可在 GitHub 上查看此文章的[示例](#)。

模型

本文中使用的模型包含单个 `Employee` 实体。

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public DateTime EmploymentStarted { get; set; }
    public int Salary { get; set; }
    public DateTime? LastPayRaise { get; set; }
}
```

在添加期间保存显式值

`Employee.EmploymentStarted` 属性配置为由数据库为新实体生成值(使用默认值)。

```
modelBuilder.Entity<Employee>()
    .Property(b => b.EmploymentStarted)
    .HasDefaultValueSql("CONVERT(date, GETDATE())");
```

以下代码可将两个员工插入到数据库中。

- 对于第一个员工，没有为 `Employee.EmploymentStarted` 属性分配任何值，因此仍将其设置为 CLR 的 `DateTime` 默认值。
- 对于第二个员工，已设置显式值 `1-Jan-2000`。

```
using (var context = new EmployeeContext())
{
    context.Employees.Add(new Employee { Name = "John Doe" });
    context.Employees.Add(new Employee { Name = "Jane Doe", EmploymentStarted = new DateTime(2000, 1, 1) });
    context.SaveChanges();

    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name + ", " + employee.EmploymentStarted);
    }
}
```

输出显示了数据库已为第一个员工生成值，且显式值已用于第二个员工。

```
1: John Doe, 1/26/2017 12:00:00 AM  
2: Jane Doe, 1/1/2000 12:00:00 AM
```

显式值插入 SQL Server IDENTITY 列

按照约定, `Employee.EmployeeId` 属性是存储生成的 `IDENTITY` 列。

对于大多数情况, 上述方法将适用于键属性。但是, 若要将显式值插入到 SQL Server `IDENTITY` 列中, 则必须在调用 `IDENTITY_INSERT` 之前手动启用 `SaveChanges()`。

NOTE

对于积压工作存在一个[功能请求](#), 请求在 SQL Server 提供程序内自动执行此操作。

```
using (var context = new EmployeeContext())  
{  
    context.Employees.Add(new Employee { EmployeeId = 100, Name = "John Doe" });  
    context.Employees.Add(new Employee { EmployeeId = 101, Name = "Jane Doe" });  
  
    context.Database.OpenConnection();  
    try  
    {  
        context.Database.ExecuteSqlRaw("SET IDENTITY_INSERT dbo.Employees ON");  
        context.SaveChanges();  
        context.Database.ExecuteSqlRaw("SET IDENTITY_INSERT dbo.Employees OFF");  
    }  
    finally  
    {  
        context.Database.CloseConnection();  
    }  
  
    foreach (var employee in context.Employees)  
    {  
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name);  
    }  
}
```

输出显示了提供的 ID 已保存到数据库。

```
100: John Doe  
101: Jane Doe
```

在更新期间设置显式值

`Employee.LastPayRaise` 属性配置为在更新期间由数据库生成值。

```
modelBuilder.Entity<Employee>()  
    .Property(b => b.LastPayRaise)  
    .ValueGeneratedOnAddOrUpdate();  
  
modelBuilder.Entity<Employee>()  
    .Property(b => b.LastPayRaise)  
    .Metadata.SetAfterSaveBehavior(PropertySaveBehavior.Ignore);
```

NOTE

默认情况下，如果尝试保存配置为在更新期间生成的属性的显式值，EF Core 将引发异常。若要避免此问题，必须下拉到较低级别的元数据 API 并设置 `AfterSaveBehavior` (如上所示)。

NOTE

EF Core 2.0 ■: 在以前版本中，通过 `IsReadOnlyAfterSave` 标志控制保存后行为。此标志已过时，将替换为 `AfterSaveBehavior`。

数据库中还存在触发器，以便在执行 `LastPayRaise` 操作期间为 `UPDATE` 列生成值。

```
CREATE TRIGGER [dbo].[Employees_UPDATE] ON [dbo].[Employees]
    AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF ((SELECT TRIGGER_NESTLEVEL()) > 1) RETURN;

    IF UPDATE(Salary) AND NOT Update(LastPayRaise)
    BEGIN
        DECLARE @Id INT
        DECLARE @OldSalary INT
        DECLARE @NewSalary INT

        SELECT @Id = INSERTED.EmployeeId, @NewSalary = Salary
        FROM INSERTED

        SELECT @OldSalary = Salary
        FROM deleted

        IF @NewSalary > @OldSalary
        BEGIN
            UPDATE dbo.Employees
            SET LastPayRaise = CONVERT(date, GETDATE())
            WHERE EmployeeId = @Id
        END
    END
END
END
```

以下代码可增加数据库中两个员工的薪金。

- 对于第一个员工，没有为 `Employee.LastPayRaise` 属性分配任何值，因此仍将设置为 null。
- 对于第二个员工，已将显式值设置为一周前(使加薪在较早的日期开始生效)。

```
using (var context = new EmployeeContext())
{
    var john = context.Employees.Single(e => e.Name == "John Doe");
    john.Salary = 200;

    var jane = context.Employees.Single(e => e.Name == "Jane Doe");
    jane.Salary = 200;
    jane.LastPayRaise = DateTime.Today.AddDays(-7);

    context.SaveChanges();

    foreach (var employee in context.Employees)
    {
        Console.WriteLine(employee.EmployeeId + ": " + employee.Name + ", " + employee.LastPayRaise);
    }
}
```

输出显示了数据库已为第一个员工生成值，且显式值已用于第二个员工。

```
1: John Doe, 1/26/2017 12:00:00 AM
2: Jane Doe, 1/19/2017 12:00:00 AM
```

EF Core 支持的 .NET 实现

2020/4/8 • [Edit Online](#)

我们希望 EF Core 可供开发人员在所有新式 .NET 实现上使用，并且我们仍在努力实现这一目标。尽管自动测试证明 EF Core 支持 .NET Core，同时许多应用程序已成功使用 EF Core，但 Mono、Xamarin 和 UWP 仍存在一些问题。

概述

下表提供了每个 .NET 实现的指南：

EF CORE	2.1 3.1
.NET Standard	2.0
.NET Core	2.0
.NET Framework ⁽¹⁾	4.7.2
Mono	5.4
Xamarin.iOS ⁽²⁾	10.14
Xamarin.Android ⁽²⁾	8.0
UWP ⁽³⁾	10.0.16299
Unity ⁽⁴⁾	2018 年 1 月

⁽¹⁾ 请参阅下面的 [.NET Framework](#) 部分。

⁽²⁾ Xamarin 存在一些问题和已知限制，这些问题和限制可能会阻止部分使用 EF Core 开发的应用程序正常运行。查看[未解决问题](#)列表，了解解决方法。

⁽³⁾ 建议使用 EF Core 2.0.1 和更高版本。安装 [.NET Core UWP 6.x 包](#)。请参阅本文的[通用 Windows 平台](#)部分。

⁽⁴⁾ Unity 存在问题和已知限制。查看[未解决问题](#)列表。

.NET Framework

面向 .NET Framework 的应用程序可能需要更改为使用 .NET Standard 库：

编辑项目文件，并确保以下条目出现在初始属性组中：

```
<AutoGenerateBindingRedirects>true</AutoGenerateBindingRedirects>
```

对于测试项目，还要确保保存在以下条目：

```
<GenerateBindingRedirectsOutputType>true</GenerateBindingRedirectsOutputType>
```

如果想要使用较旧版本的 Visual Studio, 请确保将 NuGet 客户端升级到版本 3.6.0, 以便使用 .NET Standard 2.0 库。

如果可能, 我们还建议从 NuGet packages.config 迁移到 PackageReference。请将以下属性添加到项目文件:

```
<RestoreProjectStyle>PackageReference</RestoreProjectStyle>
```

通用 Windows 平台

早期的 EF Core 和 .NET UWP 版本存在许多兼容性问题, 尤其是用于使用 .NET Native 工具链编译的应用程序时。新的 .NET UWP 版本增加了对 .NET Standard 2.0 的支持, 且包含了 .NET Native 2.0, 修复了之前报告的大多数兼容性问题。我们使用 UWP 对 EF Core 2.0.1 进行了更彻底的测试, 但测试不是自动执行的。

在 UWP 上使用 EF Core 时:

- 若要优化查询性能, 请避免在 LINQ 查询中使用匿名类型。将 UWP 应用程序部署到应用商店要求使用 .NET Native 编译应用程序。使用匿名类型的查询在 .NET Native 上性能较差。
- 若要优化 `SaveChanges()` 性能, 请使用 `ChangeTrackingStrategy.ChangingAndChangedNotifications`, 并在你的实体类型中实施 `INotifyPropertyChanged`、`INotifyPropertyChanging` 和 `INotifyCollectionChanged`。

报告问题

对于未按预期工作的任意组合, 建议在 [EF Core 问题跟踪程序](#) 中创建新问题。对于特定于 Xamarin 的问题, 请使用 [Xamarin.Android](#) 或 [Xamarin.iOS](#) 问题跟踪程序。

数据库提供程序

2020/4/8 •

Entity Framework Core 可通过名为数据库提供程序的插件库访问许多不同的数据库。

当前提供程序

IMPORTANT

EF Core 提供程序由多种源生成。并非所有提供程序均作为 Entity Framework Core 项目的组成部分进行维护。考虑使用提供程序时, 请务必评估质量、授权、支持等因素, 确保其满足要求。同时也请务必查看每个提供程序的文档, 详细了解版本兼容性信息。

IMPORTANT

EF Core 提供程序通常跨次要版本工作, 但不跨主要版本工作。例如, 针对 EF Core 2.1 发布的提供程序应用于 EF Core 2.2, 但不适用于 EF Core 3.0。

NUGET 包	数据库	状态	状态	版本	链接
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2012 以上版本	EF Core 项目 (Microsoft)		3.1	docs
Microsoft.EntityFrameworkCore.SQLite	SQLite 3.7 及以 上版本	EF Core 项目 (Microsoft)		3.1	docs
Microsoft.EntityFrameworkCore.InMemory	EF Core 内存中 数据库	EF Core 项目 (Microsoft)	限制	3.1	docs
Microsoft.EntityFrameworkCore.Cosmos	Azure Cosmos DB SQL API	EF Core 项目 (Microsoft)		3.1	docs
Npgsql.EntityFrameworkCore.PostgreSQL	postgresql	Npgsql 开发团队		3.1	docs
Pomelo.EntityFrameworkCore.MySql	MySQL、 MariaDB	Pomelo Foundation 项目		3.1	自述文件
Devart.Data.MySql.EFCore	MySQL 5 及以上 版本	DevArt	已付	3.0	docs
Devart.Data.Oracl.EFCore	Oracle DB 9.2.0.4 及更高版 本	DevArt	已付	3.0	docs

NUGET 包	功能描述	作者	状态	版本	资源
Devart.Data.PostgreSQL.EntityFrameworkCore	PostgreSQL 8.0 及以上版本	DevArt	已付	3.0	docs
Devart.Data.SQLite.EntityFrameworkCore	SQLite 3 及以上版本	DevArt	已付	3.0	docs
FileContextCore	在文件中存储数据	Morris Janatzek	用于开发	3.0	自述文件
EntityFrameworkCore.Jet	Microsoft Access 文件	Bubi	.NET Framework	2.2	自述文件
EntityFrameworkCore.SqlServerCompact35	SQL Server Compact 3.5	Erik Ejlskov Jensen	.NET Framework	2.2	wiki
EntityFrameworkCore.SqlServerCompact40	SQL Server Compact 4.0	Erik Ejlskov Jensen	.NET Framework	2.2	wiki
FirebirdSql.EntityFrameworkCore.Firebird	Firebird 2.5 和 3.x	Jiří Činčura		2.2	docs
Teradata.EntityFrameworkCore	Teradata 数据库 16.10 及更高版本	Teradata		2.2	网站
EntityFrameworkCore.FirebirdSql	Firebird 2.5 和 3.x	Rafael Almeida		2.1	wiki
EntityFrameworkCore.OpenEdge	Progress OpenEdge	Alex Wiese		2.1	自述文件
MySQL.Data.EntityFrameworkCoreCore	MySQL	MySQL 项目 (Oracle)		2.1	docs
Oracle.EntityFrameworkCore	Oracle DB 11.2 及更高版本	Oracle		2.1	网站
IBM.EntityFrameworkCore	Db2、Informix	IBM	Windows 版本	2.0	博客
IBM.EntityFrameworkCore-Inx	Db2、Informix	IBM	Linux 版本	2.0	博客
IBM.EntityFrameworkCore-osx	Db2、Informix	IBM	macOS 版本	2.0	博客
Pomelo.EntityFrameworkCore.MyCat	MyCAT 服务器	Pomelo Foundation 项目	仅预发行版	1.1	自述文件

向应用程序添加数据库提供程序

EF Core 的大多数数据库提供程序都是作为 NuGet 包分发的，可按如下所示安装：

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package provider_package_name
```

安装后，需采用 `OnConfiguring` 方法或 `AddDbContext` 方法（如果使用的是依赖关系注入容器）在 `DbContext` 中配置提供程序。例如，以下行使用传递的连接字符串配置 SQL Server 提供程序：

```
optionsBuilder.UseSqlServer(  
    "Server=(localdb)\mssqllocaldb;Database=MyDatabase;Trusted_Connection=True;");
```

数据库提供程序可扩展 EF Core，启用特定数据库特有的功能。一些概念为大多数据库共有，它们包含于 EF Core 主要组件中。此类概念包括 LINQ 中的表达查询、事务以及对象从数据库加载后的跟踪更改。另一些概念特定于具体的提供程序。例如，通过 SQL Server 提供程序可[配置内存优化表](#)（SQL Server 的一种特定功能）。其他一些概念特定于某一类提供程序。例如，用于关系数据库的 EF Core 提供程序构建于公共 `Microsoft.EntityFrameworkCore.Relational` 库上，该库提供的 API 可用于配置表和列映射、外键约束等。提供程序通常作为 NuGet 包分发。

IMPORTANT

发布 EF Core 的新补丁版本时，其中通常包含 `Microsoft.EntityFrameworkCore.Relational` 包的更新。添加关系数据库提供程序时，该包将成为应用程序的传递依赖项。但许多提供程序是独立于 EF Core 发布的，因此可能不会更新为依赖该包的较新补丁版本。为确保能修复所有 bug，建议将 `Microsoft.EntityFrameworkCore.Relational` 补丁版本添加为应用程序的直接依赖项。

Microsoft SQL Server EF Core 数据库提供程序

2020/4/8 ·

此数据库提供程序允许将 Entity Framework Core 与 Microsoft SQL Server(包括 Azure SQL 数据库)一起使用。该提供程序作为 [Entity Framework Core 项目](#) 的组成部分进行维护。

安装

安装 [Microsoft.EntityFrameworkCore.SqlServer NuGet 包](#)。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

NOTE

自版本 3.0.0 起, 提供程序引用 Microsoft.Data.SqlClient(以前的版本依赖于 SqlClient)。如果项目直接依赖于 SqlClient, 请确保它引用了 Microsoft.Data.SqlClient 包。

支持的数据库引擎

- Microsoft SQL Server(2012 及以上版本)

内存优化表支持 SQL Server EF Core 数据库提供程序

2020/3/11 •

[内存优化表](#)是 SQL Server 的一项功能，其中整个表都驻留在内存中。表数据的另一个副本维护在磁盘上，但仅用于持续性目的。在数据库恢复期间，内存优化的表中的数据只能从磁盘读取。例如，在服务器重新启动后。

配置内存优化表

你可以指定实体映射到的表是内存优化表。使用 EF Core 创建和维护基于模型的数据库（使用[迁移](#)或[EnsureCreated](#)）时，将为这些实体创建内存优化表。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().IsMemoryOptimized();
}
```

指定 Azure SQL 数据库选项

2020/3/11 •

NOTE

此 API 是 EF Core 3.1 中新增的。

Azure SQL 数据库提供了[各种定价选项](#)，这些选项通常通过 Azure 门户进行配置。但是，如果使用[EF Core 迁移](#)来管理架构，则可以在模型本身中指定所需的选项。

您可以使用[HasServiceTier](#)指定数据库的服务层：

```
modelBuilder.HasServiceTier("BusinessCritical");
```

您可以使用[HasDatabaseMaxSize](#)指定数据库的最大大小：

```
modelBuilder.HasDatabaseMaxSize("2 GB");
```

您可以使用[HasPerformanceLevel](#)指定数据库的性能级别(SERVICE_OBJECTIVE)：

```
modelBuilder.HasPerformanceLevel("BC_Gen4_1");
```

使用[HasPerformanceLevelSql](#)配置弹性池，因为该值不是字符串文本：

```
modelBuilder.HasPerformanceLevelSql("ELASTIC_POOL ( name = myelasticpool )");
```

TIP

可以在[ALTER DATABASE 文档](#)中找到所有支持的值。

SQLite EF Core 数据库提供程序

2020/4/8 ·

此数据库提供程序允许将 Entity Framework Core 与 SQLite 一起使用。该提供程序作为 Entity Framework Core 项目的组成部分进行维护。

安装

安装 [Microsoft.EntityFrameworkCore.Sqlite NuGet 包](#)。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

支持的数据库引擎

- SQLite(3.7 及以上版本)

限制

有关 SQLite 提供程序的一些重要限制, 请参阅 [SQLite 限制](#)。

SQLite EF Core 数据库提供程序限制

2020/3/11 ·

SQLite 提供程序有很多迁移限制。其中的大多数限制是由基础 SQLite 数据库引擎中的限制引起的，并不特定于 EF。

建模限制

公共关系库(由实体框架关系数据库提供程序共享)定义了用于建模大多数关系数据库引擎所共有的概念的 API。SQLite 提供程序不支持其中几个概念。

- 架构
- 序列
- 计算列

查询限制

SQLite 本身并不支持以下数据类型。EF Core 可以读取和写入这些类型的值，也支持查询相等性(`where e.Property == value`)。但其他操作(如比较和排序)将需要对客户端进行评估。

- DateTimeOffset
- Decimal
- TimeSpan
- UInt64

建议使用 DateTime 值，而不是 `DateTimeOffset`。处理多个时区时，建议在保存之前将值转换为 UTC，然后将其转换回适当的时区。

`Decimal` 类型提供了较高的精度级别。但是，如果不需要该级别的精度，则建议改为使用 `double`。您可以使用[值转换器](#)在类中继续使用 `decimal`。

```
modelBuilder.Entity<MyEntity>()
    .Property(e => e.DecimalProperty)
    .HasConversion<double>();
```

迁移限制

SQLite 数据库引擎不支持许多其他关系数据库所支持的架构操作。如果尝试将不受支持的操作之一应用于 SQLite 数据库，则会引发 `NotSupportedException`。

操作	支持？	说明
AddColumn	✓	1.0
AddForeignKey	X	
AddPrimaryKey	X	
AddUniqueConstraint	X	

迁移	支持？	版本
AlterColumn	X	
CreateIndex	✓	1.0
CreateTable	✓	1.0
DropColumn	X	
DropForeignKey	X	
Droplndex	✓	1.0
DropPrimaryKey	X	
DropTable	✓	1.0
DropUniqueConstraint	X	
RenameColumn	✓	2.2.2
RenameIndex	✓	2.1
RenameTable	✓	1.0
EnsureSchema	✓ (no-op)	2.0
DropSchema	✓ (no-op)	2.0
插入	✓	2.0
更新	✓	2.0
删除	✓	2.0

迁移限制解决方法

通过在迁移中手动编写代码来执行表重新生成，可以解决其中一些限制。表重新生成包括重命名现有表、创建新表、将数据复制到新表和删除旧表。你将需要使用 `Sql(string)` 方法来执行其中一些步骤。

有关更多详细信息，请参阅在 SQLite 文档中[进行其他类型的表架构更改](#)。

将来，EF 可以通过使用表中的 "重新生成" 方法来支持这些操作。你可以在[GitHub 项目上跟踪此功能](#)。

EF Core Azure Cosmos DB Provider

2020/4/8 ·

NOTE

此提供程序是 EF Core 3.0 新增内容。

此数据库提供程序允许将 Entity Framework Core 与 Azure Cosmos DB 一起使用。该提供程序作为 [Entity Framework Core 项目](#) 的组成部分进行维护。

在阅读本部分之前，强烈建议先熟悉 [Azure Cosmos DB 文档](#)。

NOTE

此提供程序仅适用于 Azure Cosmos DB 的 SQL API。

安装

安装 [Microsoft.EntityFrameworkCore.Cosmos NuGet 包](#)。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package Microsoft.EntityFrameworkCore.Cosmos
```

入门

TIP

可在 [GitHub 示例](#) 中查看此文章的示例。

与其他提供程序一样，第一步是调用 [UseCosmos](#)：

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder.UseCosmos(
        "https://localhost:8081",
        "C2y6yDjf5/R+ob0N8A7Cgv30VRDJWEHLM+4QDU5DE2nQ9nDuVTqobD4b8mGGyPMbIZnqyMsEcaGQy67XIw/Jw==",
        databaseName: "OrdersDB");
```

WARNING

为了简单起见，此处对终结点和密钥进行了硬编码，但在生产应用中，应[安全地存储](#)这些终结点和密钥。

在本例中，`Order` 是一个简单实体，其中包含对从属类型 `StreetAddress` 的引用。

```
public class Order
{
    public int Id { get; set; }
    public int? TrackingNumber { get; set; }
    public string PartitionKey { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}
```

```
public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

保存和查询数据遵循常规 EF 模式：

```
using (var context = new OrderContext())
{
    await context.Database.EnsureDeletedAsync();
    await context.Database.EnsureCreatedAsync();

    context.Add(new Order
    {
        Id = 1,
        ShippingAddress = new StreetAddress { City = "London", Street = "221 B Baker St" },
        PartitionKey = "1"
    });

    await context.SaveChangesAsync();
}

using (var context = new OrderContext())
{
    var order = await context.Orders.FirstAsync();
    Console.WriteLine($"First order will ship to: {order.ShippingAddress.Street},
{order.ShippingAddress.City}");
    Console.WriteLine();
}
```

IMPORTANT

要创建所需的容器并插入种子数据(如果存在于模型中)，则需要调用 `EnsureCreatedAsync`。但是只应在部署期间调用 `EnsureCreatedAsync`，而不应在正常操作中调用，否则可能会导致性能问题。

特定于 Cosmos 的模型自定义

默认情况下，所有实体类型都映射到同一个容器，该容器以派生的上下文命名(在本例中为 `"OrderContext"`)。要更改默认容器名称，请使用 `HasDefaultContainer`：

```
modelBuilder.HasDefaultContainer("Store");
```

要将实体类型映射到其他容器，请使用 `ToContainer`：

```
modelBuilder.Entity<Order>()
    .ToContainer("Orders");
```

为了标识给定项表示的实体类型，EF Core 添加鉴别器值（即使没有派生实体类型）。[可以更改](#) 鉴别器的名称和值。

如果其他实体类型永远不会存储在同一个容器中，则可以通过调用 [HasNoDiscriminator](#) 删除鉴别器：

```
modelBuilder.Entity<Order>()
    .HasNoDiscriminator();
```

分区键

默认情况下，EF Core 将创建分区键设置为 `"__partitionKey"` 的容器，而不会在插入项时为其提供任何值。但若要充分利用 Azure Cosmos 的性能功能，[应仔细选择应使用的分区键](#)。可以通过调用 [HasPartitionKey](#) 来配置它：

```
modelBuilder.Entity<Order>()
    .HasPartitionKey(o => o.PartitionKey);
```

NOTE

只要分区键属性[转换为字符串](#)，则它可以为任意类型。

配置分区键属性后，应始终具有非 null 值。发出查询时，可以添加条件将其设置为单分区。

```
using (var context = new OrderContext())
{
    context.Add(new Order
    {
        Id = 2,
        ShippingAddress = new StreetAddress { City = "New York", Street = "11 Wall Street" },
        PartitionKey = "2"
    });

    await context.SaveChangesAsync();
}

using (var context = new OrderContext())
{
    var order = await context.Orders.Where(p => p.PartitionKey == "2").LastAsync();
    Console.WriteLine("Last order will ship to: ");
    Console.WriteLine($"{order.ShippingAddress.Street}, {order.ShippingAddress.City}");
    Console.WriteLine();
}
```

嵌入的实体

对于 Cosmos，从属实体嵌入到所有者所在的项中。要更改属性名称，请使用 [ToJsonProperty](#)：

```
modelBuilder.Entity<Order>().OwnsOne(
    o => o.ShippingAddress,
    sa =>
    {
        sa.ToJsonProperty("Address");
        sa.Property(p => p.Street).ToJsonProperty("ShipsToStreet");
        sa.Property(p => p.City).ToJsonProperty("ShipsToCity");
    });
});
```

对于此配置，以上示例中的顺序存储如下：

```
{
    "Id": 1,
    "PartitionKey": "1",
    "TrackingNumber": null,
    "id": "1",
    "Address": {
        "ShipsToCity": "London",
        "ShipsToStreet": "221 B Baker St"
    },
    "_rid": "6QEKAM+BOOABAAAAAAA==",
    "_self": "dbs/6QEKAA=/colls/6QEKAM+BOOA=/docs/6QEKAM+BOOABAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-683c-692e763901d5\"",
    "_attachments": "attachments/",
    "_ts": 1568163674
}
```

还嵌入了从属实体的集合。对于下一个示例，我们将使用具有 `Distributor` 集合的 `StreetAddress` 类：

```
public class Distributor
{
    public int Id { get; set; }
    public ICollection<StreetAddress> ShippingCenters { get; set; }
}
```

从属实体不需要提供要存储的显式键值：

```
var distributor = new Distributor
{
    Id = 1,
    ShippingCenters = new HashSet<StreetAddress> {
        new StreetAddress { City = "Phoenix", Street = "500 S 48th Street" },
        new StreetAddress { City = "Anaheim", Street = "5650 Dolly Ave" }
    }
};

using (var context = new OrderContext())
{
    context.Add(distributor);

    await context.SaveChangesAsync();
}
```

它们将以这种方式持久保存：

```
{
    "Id": 1,
    "Discriminator": "Distributor",
    "id": "Distributor|1",
    "ShippingCenters": [
        {
            "City": "Phoenix",
            "Street": "500 S 48th Street"
        },
        {
            "City": "Anaheim",
            "Street": "5650 Dolly Ave"
        }
    ],
    "_rid": "6QEKANzISj0BAAAAAAA==",
    "_self": "dbs/6QEKA==/colls/6QEKANzISj0=/docs/6QEKANzISj0BAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-683c-7b2b439701d5\"",
    "_attachments": "attachments/",
    "_ts": 1568163705
}
```

在内部而言，EF Core 始终需要对所有被跟踪实体提供唯一键值。默认情况下，为从属类型集合创建的主键包含指向所有者的外键属性和与 JSON 数组中的索引对应的 `int` 属性。要检索这些值，可使用以下条目 API：

```
using (var context = new OrderContext())
{
    var firstDistributor = await context.Distributors.FirstAsync();
    Console.WriteLine($"Number of shipping centers: {firstDistributor.ShippingCenters.Count}");

    var addressEntry = context.Entry(firstDistributor.ShippingCenters.First());
    var addressPKProperties = addressEntry.Metadata.FindPrimaryKey().Properties;

    Console.WriteLine($"First shipping center PK:
({addressEntry.Property(addressPKProperties[0].Name).CurrentValue},
{addressEntry.Property(addressPKProperties[1].Name).CurrentValue})");
    Console.WriteLine();
}
```

TIP

必要时，可更改从属实体类型的默认主键，但应显式提供键值。

使用断开连接的实体

每个项都需要具有一个对于给定分区键唯一的 `id` 值。默认情况下 EF Core 通过使用 ‘|’ 作为分隔符串联鉴别器和主键值来生成值。仅当实体进入 `Added` 状态时才生成键值。如果附加实体在 .NET 类型上没有用于存储值的 `id` 属性，则这可能会导致问题。

要解决此限制，可以手动创建并设置 `id` 值，或者先将实体标记为已添加，然后将其更改为所需状态：

```

using (var context = new OrderContext())
{
    var distributorEntry = context.Add(distributor);
    distributorEntry.State = EntityState.Unchanged;

    distributor.ShippingCenters.Remove(distributor.ShippingCenters.Last());

    await context.SaveChangesAsync();
}

using (var context = new OrderContext())
{
    var firstDistributor = await context.Distributors.FirstAsync();
    Console.WriteLine($"Number of shipping centers is now: {firstDistributor.ShippingCenters.Count}");

    var distributorEntry = context.Entry(firstDistributor);
    var idProperty = distributorEntry.Property<string>("id");
    Console.WriteLine($"The distributor 'id' is: {idProperty.CurrentValue}");
}

```

生成的 JSON 如下：

```
{
    "Id": 1,
    "Discriminator": "Distributor",
    "id": "Distributor|1",
    "ShippingCenters": [
        {
            "City": "Phoenix",
            "Street": "500 S 48th Street"
        }
    ],
    "_rid": "JBwtAN8oNYEBAAAAAAAA==",
    "_self": "dbs/JBwtAA==/colls/JBwtAN8oNYE=/docs/JBwtAN8oNYEBAAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-9377-d7a1ae7c01d5\"",
    "_attachments": "attachments/",
    "_ts": 1572917100
}
```

使用 EF Core Azure Cosmos DB 提供程序中的非结构化数据

2020/3/11 •

EF Core 旨在使使用在模型中定义的架构的数据变得简单。但 Azure Cosmos DB 的优点之一是存储数据形状的灵活性。

访问原始 JSON

可以通过名为 `"__jObject"` 的[卷影状态](#)中 `JObject` 包含表示从存储区接收的数据的特定属性来访问不 EF Core 跟踪的属性，以及将存储的数据：

```
using (var context = new OrderContext())
{
    await context.Database.EnsureDeletedAsync();
    await context.Database.EnsureCreatedAsync();

    var order = new Order
    {
        Id = 1,
        ShippingAddress = new StreetAddress { City = "London", Street = "221 B Baker St" },
        PartitionKey = "1"
    };

    context.Add(order);

    await context.SaveChangesAsync();
}

using (var context = new OrderContext())
{
    var order = await context.Orders.FirstAsync();
    var orderEntry = context.Entry(order);

    var jsonProperty = orderEntry.Property< JObject>("__jObject");
    jsonProperty.CurrentValue["BillingAddress"] = "Clarence House";

    orderEntry.State = EntityState.Modified;

    await context.SaveChangesAsync();
}

using (var context = new OrderContext())
{
    var order = await context.Orders.FirstAsync();
    var orderEntry = context.Entry(order);
    var jsonProperty = orderEntry.Property< JObject>("__jObject");

    Console.WriteLine($"First order will be billed to: {jsonProperty.CurrentValue["BillingAddress"]}");
}
```

```
{  
    "Id": 1,  
    "PartitionKey": "1",  
    "TrackingNumber": null,  
    "id": "1",  
    "Address": {  
        "ShipsToCity": "London",  
        "ShipsToStreet": "221 B Baker St"  
    },  
    "_rid": "eLMaAK8TzkIBAAAAAAA==",  
    "_self": "dbs/eLMaAA==/colls/eLMaAK8TzkI=/docs/eLMaAK8TzkIBAAAAAAA==/",  
    "_etag": "\"00000000-0000-0000-683e-0a12bf8d01d5\"",  
    "_attachments": "attachments/",  
    "BillingAddress": "Clarence House",  
    "_ts": 1568164374  
}
```

WARNING

`__jObject` 属性是 EF Core 基础结构的一部分，只应用作最后的手段，因为在将来的版本中可能会有不同的行为。

NOTE

对实体所做的更改将覆盖 `SaveChanges` 期间 `__jObject` 中存储的值。

使用 CosmosClient

若要完全分离 EF Core 获取 `DbContext` 中 [AZURE COSMOS DB SDK 的一部分](#) 的 `CosmosClient` 对象：

```
using (var context = new OrderContext())  
{  
    var cosmosClient = context.Database.GetCosmosClient();  
    var database = cosmosClient.GetDatabase("OrdersDB");  
    var container = database.GetContainer("Orders");  
  
    var resultSet = container.GetItemQueryIterator<JObject>(new QueryDefinition("select * from o"));  
    var order = (await resultSet.ReadNextAsync()).First();  
  
    Console.WriteLine($"First order JSON: {order}");  
  
    order.Remove("TrackingNumber");  
  
    await container.ReplaceItemAsync(order, order["id"].ToString());  
}
```

缺少属性值

在上面的示例中，我们从顺序中删除了 `"TrackingNumber"` 属性。由于在 Cosmos DB 中索引的工作原理，引用缺少的属性的其他位置的查询可能会返回意外的结果。例如：

```
using (var context = new OrderContext())
{
    var orders = await context.Orders.ToListAsync();
    var sortedOrders = await context.Orders.OrderBy(o => o.TrackingNumber).ToListAsync();

    Console.WriteLine($"Number of orders: {orders.Count}");
    Console.WriteLine($"Number of sorted orders: {sortedOrders.Count}");
}
```

排序的查询实际上不返回任何结果。这意味着，当直接使用存储时，应注意始终填充由 EF Core 映射的属性。

NOTE

在未来版本的 Cosmos 中，此行为可能会发生变化。例如，当前如果索引策略定义了复合索引 {Id/? ASC, TrackingNumber/? ASC}，则为 "ORDER BY c.Id ASC, ASC" 的查询将返回缺少 "TrackingNumber" 属性的项。

Azure Cosmos DB 提供程序限制 EF Core

2020/3/11 •

Cosmos 提供程序有很多限制。其中的许多限制是由基础 Cosmos 数据库引擎中的限制引起的，并不特定于 EF。但大多数情况还[没有实现](#)。

临时限制

- 即使只有一个实体类型没有映射到容器，它仍具有鉴别器属性。
- 在某些情况下，具有分区键的实体类型不能正常工作
- 不支持 `Include` 调用
- 不支持 `Join` 调用

Azure Cosmos DB SDK 限制

- 仅提供 `async` 方法

WARNING

由于没有 EF Core 依赖的低级方法的同步版本，因此当前通过对返回的 `Task` 调用 `.Wait()` 来实现相应的功能。这意味着使用 `SaveChanges` 或 `ToList` 等方法，而不是其异步对应项可能导致应用程序中出现死锁

Azure Cosmos DB 限制

你可以查看[Azure Cosmos DB 支持的功能](#)的完整概述，与关系数据库相比，这些是最明显的区别：

- 不支持客户端启动的事务
- 一些跨分区查询要么不受支持，要么速度慢得多，具体取决于所涉及的运算符

EF Core In-Memory 数据库提供程序

2020/4/8 ·

此数据库提供程序允许将 Entity Framework Core 和内存数据库一起使用。这对测试非常有用，尽管内存中模式下的 SQLite 提供程序可能是针对关系数据库的更合适的测试替代。该提供程序作为 [Entity Framework Core 项目](#) 的组成部分进行维护。

安装

安装 [Microsoft.EntityFrameworkCore.InMemory NuGet 包](#)。

- [.NET Core CLI](#)
- [Visual Studio](#)

```
dotnet add package Microsoft.EntityFrameworkCore.InMemory
```

入门

下列资源可帮助你开始使用此提供程序。

- [使用 InMemory 进行测试](#)
- [UnicornStore 示例应用程序测试](#)

支持的数据库引擎

进程中内存数据库(仅用于测试目的)

编写数据库提供程序

2020/3/11 •

有关编写 Entity Framework Core 数据库提供程序的信息，请参阅，希望通过[Arthur Vickers编写 EF Core 提供程序](#)。

NOTE

自 EF Core 1.1 起，这些文章尚未更新，因为此时间之后发生了重大更改，[681](#)正在跟踪此文档的更新。

EF Core 基本代码是开放源代码，并且包含多个可用作参考的数据库提供程序。可以在<https://github.com/aspnet/EntityFrameworkCore>中找到源代码。查看常用的第三方提供程序（例如[Npgsql](#)、[Pomelo MySQL](#)和[SQL Server Compact](#)）的代码可能也很有帮助。特别是，设置这些项目是为了从和运行我们在 NuGet 上发布的功能测试。强烈建议使用这种设置。

提供提供程序更改的最新状态

从2.1 版本开始，开始使用后，我们创建了一个可能需要对提供程序代码进行相应更改的[更改日志](#)。这旨在帮助更新现有提供程序以使用 EF Core 的新版本。

在2.1 之前，我们使用了 `providers-beware`，并 `providers-fyi` 了 GitHub 问题的标签和拉取请求来实现类似目的。我们将 `continues` 将这些标签用于问题，指出给定版本中的哪些工作项可能还需要在提供程序中完成工作。

`providers-beware` 标签通常意味着，工作项的实现可能会中断提供程序，而 `providers-fyi` 标签通常意味着提供程序将不会中断，但可能仍需要更改代码，例如，启用新功能。

建议的第三方提供程序命名

建议为 NuGet 包使用以下命名。这是与 EF Core 小组所传递的包名称一致。

```
<Optional project/company name>.EntityFrameworkCore.<Database engine name>
```

例如：

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Npgsql.EntityFrameworkCore.PostgreSQL`
- `EntityFrameworkCore.SqlServerCompact40`

提供程序影响的更改

2020/3/11 •

此页包含的链接指向在 EF Core 存储库上发出的请求，这些请求可能需要其他数据库提供程序的作者做出反应。目的是在将提供程序更新为新版本时，为现有的第三方数据库提供程序的作者提供一个起点。

我们正在从 2.1 到 2.2 的更改启动此日志。在 2.1 之前，我们使用 `providers-beware`，并 `providers-fyi` 标签和拉取请求。

2.2 ---> 3.0

请注意，许多[应用程序级别的重大更改](#)还会影响提供程序。

- <https://github.com/aspnet/EntityFrameworkCore/pull/14022>
 - 删除了过时的 API 和折叠的可选参数重载
 - 已删除 `DatabaseColumn.GetUnderlyingStoreType()`
- <https://github.com/aspnet/EntityFrameworkCore/pull/14589>
 - 删除过时的 API
- <https://github.com/aspnet/EntityFrameworkCore/pull/15044>
 - 由于在基实现中修复几个 bug 所需的行为更改，`CharTypeMapping` 的子类可能已中断。
- <https://github.com/aspnet/EntityFrameworkCore/pull/15090>
 - 添加了 `IDatabaseModelFactory` 的基类，并将其更新为使用参数对象来缓解将来的中断。
- <https://github.com/aspnet/EntityFrameworkCore/pull/15123>
 - 在 `MigrationsSqlGenerator` 中使用的参数对象可缓解未来的中断。
- <https://github.com/aspnet/EntityFrameworkCore/pull/14972>
 - 日志级别的显式配置需要对提供程序可能使用的 API 进行一些更改。具体而言，如果提供程序直接使用日志记录基础结构，此更改可能会破坏使用。此外，使用基础结构（将是公共的）的提供程序将需要从 `LoggingDefinitions` 或 `RelationalLoggingDefinitions` 派生。有关示例，请参阅 SQL Server 和内存中提供程序。
- <https://github.com/aspnet/EntityFrameworkCore/pull/15091>
 - 核心、关系和抽象资源字符串现在是公共的。
 - `CoreLoggerExtensions` 和 `RelationalLoggerExtensions` 现在是公共的。当记录在核心或关系级别定义的事件时，提供程序应使用这些 API。不要直接访问日志记录资源；这些都是内部的。
 - `IRawSqlCommandBuilder` 已从单一服务更改为范围服务
 - `IMigrationsSqlGenerator` 已从单一服务更改为范围服务
- <https://github.com/aspnet/EntityFrameworkCore/pull/14706>
 - 构建关系命令的基础结构已成为公共的，以便提供程序可以安全地使用它，并对其进行略微重构。
- <https://github.com/aspnet/EntityFrameworkCore/pull/14733>
 - `ILazyLoader` 已从作用域服务更改为暂时性服务
- <https://github.com/aspnet/EntityFrameworkCore/pull/14610>
 - `IUpdateSqlGenerator` 已从作用域服务更改为单一服务
 - 此外，`ISingletonUpdateSqlGenerator` 已被删除
- <https://github.com/aspnet/EntityFrameworkCore/pull/15067>
 - 提供商现在使用的大量内部代码已公开
 - 它不应再 necessary 引用 `IndentedStringBuilder` 因为它已被从公开它的位置中分离出来
 - 应将 `NonCapturingLazyInitializer` 的用法替换为 BCL `LazyInitializer`

- <https://github.com/aspnet/EntityFrameworkCore/pull/14608>
 - 应用程序重大更改文档完全涵盖了此更改。对于提供程序，这可能会产生更大的影响，因为测试 EF core 常常会导致此问题，因此测试基础结构发生了变化以降低可能性。
- <https://github.com/aspnet/EntityFrameworkCore/issues/13961>
 - 已简化 `EntityMaterializerSource`
- <https://github.com/aspnet/EntityFrameworkCore/pull/14895>
 - StartsWith 翻译以提供商可能需要/需要反应的方式进行了更改
- <https://github.com/aspnet/EntityFrameworkCore/pull/15168>
 - 约定集服务已更改。提供程序现在应继承自 "ProviderConventionSet" 或 "RelationalConventionSet"。
 - 可以通过 `IConventionSetCustomizer` 服务添加自定义，但这种方法旨在供其他扩展使用，而不是由提供程序使用。
 - 应从 `IConventionSetBuilder` 解析运行时使用的约定。
- <https://github.com/aspnet/EntityFrameworkCore/pull/15288>
 - 数据种子已重构为公共 API，以避免需要使用内部类型。这将会影响非关系提供程序，因为种子设定由所有关系提供程序的基本关系类处理。

2.1 ---> 2.2

仅限测试更改

- <https://github.com/aspnet/EntityFrameworkCore/pull/12057>-允许在测试中自定义 SQL 分隔符
 - 在 `BuiltInDataTypesTestBase` 中允许非严格浮点比较的测试更改
 - 允许将查询测试与不同的 SQL 分隔符重新使用的测试更改
- <https://github.com/aspnet/EntityFrameworkCore/pull/12072>-向关系规范测试添加 `DbFunction` 测试
 - 这些测试可以对所有数据库提供程序运行
- <https://github.com/aspnet/EntityFrameworkCore/pull/12362>-异步测试清理
 - 删除 `Wait` 调用、不需要的异步，并重命名了一些测试方法
- <https://github.com/aspnet/EntityFrameworkCore/pull/12666>-统一日志测试基础结构
 - 添加了 `CreateListLoggerFactory` 并删除了一些以前的日志结构，这将需要使用这些测试的提供程序来做出响应
- <https://github.com/aspnet/EntityFrameworkCore/pull/12500>-同步和异步运行多个查询测试
 - 测试名称和因式分解已更改，这将要求使用这些测试的提供程序做出反应
- 在 ComplexNavigations 模型中重命名导航 <https://github.com/aspnet/EntityFrameworkCore/pull/12766>
 - 使用这些测试的提供程序可能需要响应
- <https://github.com/aspnet/EntityFrameworkCore/pull/12141>-将上下文返回到池，而不是在功能测试中释放
 - 此更改包括一些可能要求提供程序做出反应的测试重构

测试和产品代码更改

- <https://github.com/aspnet/EntityFrameworkCore/pull/12109>-合并 `RelationalTypeMapping` 方法
 - 在派生类中，简化了 2.1 到 `RelationalTypeMapping` 的更改。我们认为这是不会中断提供程序的，但提供程序可以在其派生的类型映射类中利用这一更改。
- <https://github.com/aspnet/EntityFrameworkCore/pull/12069> 标记的查询或命名查询
 - 添加用于标记 LINQ 查询的基础结构，并使这些标记在 SQL 中显示为注释。这可能需要提供程序在 SQL 生成中做出反应。
- <https://github.com/aspnet/EntityFrameworkCore/pull/13115>-通过 NTS 支持空间数据
 - 允许在提供程序外部注册类型映射和成员转换器
 - 提供程序必须调用 `base.ITypeMappingSource` 实现中的 `FindMapping()`
 - 按照此模式将空间支持添加到在提供程序中保持一致的提供程序。
- <https://github.com/aspnet/EntityFrameworkCore/pull/13199>-添加用于服务提供程序创建的增强调试

- 允许 DbContextOptionsExtensions 实现一个新的接口，该接口可帮助用户了解如何重新生成内部服务提供程序
- <https://github.com/aspnet/EntityFrameworkCore/pull/13289>-添加用于运行状况检查的后 API
 - 此 PR 添加了 `CanConnect` 的概念，这些概念将由 ASP.NET Core 运行状况检查用来确定数据库是否可用。默认情况下，关系实现仅调用 `Exist`，但如果有必要，提供程序可实现不同的操作。非关系提供程序将需要实现新的 API，才能使用运行状况检查。
- <https://github.com/aspnet/EntityFrameworkCore/pull/13306>-Update base RelationalTypeMapping 不设置 DbParameter 大小
 - 默认情况下，停止设置大小，因为它会导致截断。如果需要设置大小，则提供程序可能需要添加自己的逻辑。
- <https://github.com/aspnet/EntityFrameworkCore/pull/13372> RevEng: 始终为 decimal 列指定列类型
 - 始终在基架代码中为十进制列配置列类型，而不是按约定配置列类型。
 - 提供程序不应在其端上进行任何更改。
- <https://github.com/aspnet/EntityFrameworkCore/pull/13469>-添加用于生成 SQL CASE 表达式的 CaseExpression
- <https://github.com/aspnet/EntityFrameworkCore/pull/13648>-可以在 SqlFunctionExpression 上指定类型映射，以改善参数和结果的存储类型推理。

EF Core 工具和扩展

2020/4/9 • [Edit Online](#)

这些工具和扩展为 Framework Core 2.1 及更高版本提供了附加功能。

IMPORTANT

扩展由各种源构建，不作为 Entity Framework Core 项目的一部分进行维护。考虑使用第三方扩展时，请务必评估其质量、授权、兼容性和支持等因素，确保其符合要求。具体而言，为更早版本的 EF Core 构建的扩展可能需要更新，然后才适用于最新版本。

工具

LLBLGen Pro

LLBLGen Pro 是一种实体建模解决方案，包含对 Entity Framework 和 Entity Framework Core 的支持。借助它可轻松通过 Database First 或 Model First 定义实体模型并将其映射到数据库中，使你可以立即开始编写查询。对于 EF Core:2.

[网站](#)

Devart Entity Developer

Devart Entity Developer 是一种用于 ADO.NET 实体框架、NHibernate、LinqConnect、Telerik 数据访问和 LINQ to SQL 的强大 ORM 设计器。它支持 EF Core 模型的直观设计、使用“模型优先”或“数据库优先”的方式，还支持 C# 或 Visual Basic 代码生成。对于 EF Core:2.

[网站](#)

用于 Entity Framework 的 nHydrate ORM

为 Entity Framework 创建强类型的可扩展类的 ORM。生成的代码为 Entity Framework Core。二者没有任何区别。这不能替代 EF 或自定义 ORM。它是一种视觉对象建模层，可让团队管理复杂的数据库架构。它适用于 Git 等 SCM 软件，允许多用户访问你的模型，并最大限度减少冲突。安装程序可跟踪模型更改并创建升级脚本。对于 EF Core:3.

[Github 站点](#)

EF Core Power Tools

EF Core Power Tools 是一种 Visual Studio 扩展，它在简单用户界面中公开各种 EF Core 设计时任务。其中包括对现有数据库和 [SQL Server DACPAC](#) 中的 DbContext 和实体类的反向工程、对数据库迁移的管理，以及模型可视化效果。对于 EF Core:2、3。

[GitHub wiki](#)

实体框架可视化编辑器

Entity Framework Visual Editor 是一种 Visual Studio 扩展，其中增添了 ORM 设计器用于 EF 6 和 EF Core 类的可视化设计。代码是通过 T4 模板生成的，因此可自定义来满足任意需求。它支持继承、单向和双向关联，支持枚举，还能用颜色标识类并添加文本块来解释潜在不可预测的设计部分。对于 EF Core:2.

[市场](#)

CatFactory

CatFactory 是一种面向 .NET Core 的基架引擎，它可自动基于 SQL Server 数据库生成 DbContext 类、实体、映射配

置和存储库类。对于 EF Core:2.

[GitHub 存储库](#)

LoreSoft 的 Entity Framework Core 生成器

Entity Framework Core Generator (efg) 是一种 .NET Core CLI 工具，可基于现有数据库生成 EF Core 模型，其功能与 `dotnet ef dbcontext scaffold` 很相似，但它还支持通过区域替换或分析映射文件来实现安全代码的重新生成。此工具支持生成视图模型、验证和对象映射器代码。对于 EF Core:2.

[教程 文档](#)

扩展

Microsoft.EntityFrameworkCore.AutoHistory

一个插件库，它可用于将 EF Core 执行的数据更改自动记录到历史记录表中。对于 EF Core:2.

[GitHub 存储库](#)

EFSecondLevelCache.Core

一个扩展，它可将 EF Core 查询的结果存储到二级缓存中，使后续执行相同查询时无需访问数据库，而是直接从缓存中检索数据。对于 EF Core:2.

[GitHub 存储库](#)

Geco

Geco(生成器控制台)是一个基于控制台项目的简单代码生成器，它在 .NET Core 上运行并使用 C# 内插字符串来生成代码。Geco 提供面向 EF Core 的反向模型生成器，并支持复数形式、单数形式和可编辑的模板。它还支持种子数据脚本生成器、脚本运行器和数据库清理器。对于 EF Core:2.

[GitHub 存储库](#)

EntityFrameworkCore.Scaffolding.Handlebars

允许结合使用 Entity Framework Core 工具链和 Handlebars 模板对基于现有数据库反向工程处理的类进行自定义。对于 EF Core:2、3。

[GitHub 存储库](#)

NeinLinq.EntityFrameworkCore

NeinLinq 扩展了 Entity Framework 等 LINQ 提供程序，让用户能够使用可转换谓词和选择器重复使用函数、重新编写查询并构建动态查询。对于 EF Core:2、3。

[GitHub 存储库](#)

Microsoft.EntityFrameworkCore.UnitOfWork

Microsoft.EntityFrameworkCore 的一个插件，它支持存储库、工作模式单元，并支持多个具有具有分布式事务的数据库。对于 EF Core:2.

[GitHub 存储库](#)

EFCore.BulkExtensions

用于批量操作(插入、更新和删除)的 EF Core 插件。对于 EF Core:2、3。

[GitHub 存储库](#)

Bricelam.EntityFrameworkCore.Pluralizer

添加设计时复数形式。对于 EF Core:2.

[GitHub 存储库](#)

Toolbelt.EntityFrameworkCore.IndexAttribute

恢复 [Index] 属性(带有用于模型构建的扩展)。对于 EF Core:2、3。

[GitHub 存储库](#)

EfCore.InMemoryHelpers

提供一个面向 EF Core 内存中数据库提供程序的包装器。使其功能与关系提供程序更类似。对于 EF Core:2.

[GitHub 存储库](#)

EFCore.TemporalSupport

对时态支持的实现。对于 EF Core:2.

[GitHub 存储库](#)

EfCoreTemporalTable

使用下列引入的扩展方法对你喜爱的数据库轻松执行时态查询:`AsTemporalAll()`、`AsTemporalAsOf(date)`、
`AsTemporalFrom(startDate, endDate)`、`AsTemporalBetween(startDate, endDate)`、
`AsTemporalContained(startDate, endDate)`。对于 EF Core:3.

[GitHub 存储库](#)

EFCore.TimeTraveler

允许使用你已定义的 EF Core 代码、实体和映射对 [SQL Server 时态历史记录](#) 执行功能齐全的 Entity Framework Core 查询。通过将代码包装到 `using (TemporalQuery.AsOf(targetDateTime)) {...}` 中按时间顺序查看。对于 EF Core:3.

[GitHub 存储库](#)

EntityFrameworkCore.TemporalTables

适用于 Entity Framework Core 的扩展库, 使用 SQL Server 的开发人员可通过它轻松使用时态表。对于 EF Core:2.

[GitHub 存储库](#)

EntityFrameworkCore.Cacheable

高性能二级查询缓存。对于 EF Core:2.

[GitHub 存储库](#)

Entity Framework Plus

扩展 DbContext 的功能, 例如:包括筛选器、审核、缓存、查询未来、成批删除、批量更新等。对于 EF Core:2、3。

[网站](#) [GitHub 存储库](#)

实体框架扩展

通过高性能批量操作扩展 DbContext:`BulkSaveChanges`、`BulkInsert`、`BulkUpdate`、`BulkDelete`、`BulkMerge` 等。对于 EF Core:2、3。

[网站](#)

Expressionify

添加了对在 linq lambda 中调用扩展方法的支持。对于 EF Core:3.1

[GitHub 存储库](#)

XLinq

适用于关系数据库的语言集成查询 (LINQ) 技术。它允许你使用 C# 编写强类型查询。对于 EF Core:3.1

- 完全支持使用 C# 创建查询:可在 lambda 表达式内使用多个语句, 还可使用变量、函数等。

- 与 SQL 之间不存在语义缺口。XLinq 将 SQL 语句(如 `SELECT`、`FROM`、`WHERE`)声明为第一类 C# 方法，将熟悉的语法与 intellisense、类型安全性和重构结合起来。

因此，SQL 成为了“又一个”本地公开其 API 的类库，可以说是“集成了语言的 SQL”。

[网站](#)

Entity Framework Core 工具参考

2020/4/8 ·

Entity Framework Core 工具可帮助完成设计时的开发任务。它们主要用于通过对数据库架构进行反向工程来管理迁移和搭建 `DbContext` 和实体类型的基架。

- 在 Visual Studio 中, [EF Core 程序包管理器控制台工具](#) 在[程序包管理器控制台](#)中运行。
- [EF Core.NET 命令行接口 \(CLI\) 工具](#)是对跨平台 [.NET Core CLI 工具](#)的扩展。这些工具需要 .NET Core SDK 项目(具有 `Sdk="Microsoft.NET.Sdk"` 的项目或项目文件中的相似项目)。

这两种工具提供相同的功能。如果使用 Visual Studio 进行开发, 我们建议使用[程序包管理器控制台工具](#), 因为这些工具提供更完整的体验。

后续步骤

- [EF Core 程序包管理器控制台工具参考](#)
- [EF Core.NET CLI 工具参考](#)

Entity Framework Core 工具参考 - Visual Studio 中的包管理器控制台

2020/3/11 •

适用于 Entity Framework Core 的包管理器控制台 (PMC) 工具执行设计时开发任务。例如，他们基于现有数据库创建迁移、应用迁移和生成模型的代码。命令使用[包管理器控制台](#)在 Visual Studio 中运行。这些工具同时适用于 .NET Framework 和 .NET Core 项目。

如果未使用 Visual Studio，则建议改为使用[EF Core 命令行工具](#)。CLI 工具是跨平台的，在命令提示符下运行。

安装工具

用于安装和更新这些工具的过程在 ASP.NET Core 2.1 及更早版本或其他项目类型之间有所不同。

ASP.NET Core 版本 2.1 及更高版本

工具将自动包含在 ASP.NET Core 2.1+ 项目中，因为 `Microsoft.EntityFrameworkCore.Tools` 包包含在[AspNetCore 元包](#)中。

因此，您无需执行任何操作来安装这些工具，但必须执行以下操作：

- 在新项目中使用这些工具之前还原包。
- 安装包以将工具更新到较新版本。

为了确保获得最新版本的工具，我们建议您也执行以下步骤：

- 编辑 `.csproj` 文件，并添加一行以指定最新版本的 `microsoft.entityframeworkcore` 包。例如，`.csproj` 文件可能包含如下所示的 `ItemGroup`：

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.1.3" />
  <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="2.1.1" />
</ItemGroup>
```

在收到类似于以下示例的消息时更新工具：

EF Core 工具版本 "2.1.1-30846" 早于运行时 "2.1.3-32065"。更新工具以获取最新功能和 bug 修复。

更新工具：

- 安装最新 .NET Core SDK。
- 将 Visual Studio 更新到最新版本。
- 编辑 `.csproj` 文件，使其包含对最新工具包的包引用，如前文所述。

其他版本和项目类型

在 Package Manager console 中运行以下命令，安装包管理器控制台工具：

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

在 Package Manager Console 中运行以下命令，更新这些工具。

```
Update-Package Microsoft.EntityFrameworkCore.Tools
```

验证安装

通过运行以下命令验证是否已安装这些工具：

```
Get-Help about_EntityFrameworkCore
```

输出如下所示(不告诉您正在使用的工具的版本)：

```
          _/\_
        ---=/   \
        |.   \|\
        |_||_|  | )  \\\
        |||_|  \/_| //||\
        |||_|      /  \\\|\|
```

TOPIC
about_EntityFrameworkCore

SHORT DESCRIPTION
Provides information about the Entity Framework Core Package Manager Console Tools.

<A list of available commands follows, omitted here.>

使用工具

使用这些工具之前：

- 了解目标项目和启动项目之间的差异。
- 了解如何将工具与 .NET Standard 类库一起使用。
- 对于 ASP.NET Core 项目，请设置环境。

目标和启动项目

命令引用 [项目](#) 和 [启动项目](#)。

- 该 [项目](#) 也称为 [目标项目](#)，因为它是命令在其中添加或删除文件的位置。默认情况下，在“[程序包管理器控制台](#)”中选择的 [默认项目](#) 是目标项目。您可以使用 `--project` 选项指定其他项目作为目标项目。
- [启动项目](#) 是工具生成和运行的项目。这些工具必须在设计时执行应用程序代码，以获取有关项目的信息，例如数据库连接字符串和模型的配置。默认情况下，[解决方案资源管理器](#) 中的启动项目是启动项目。您可以使用“`--startup-project`”选项指定一个不同的项目作为启动项目。

启动项目和目标项目通常是同一个项目。它们是不同的项目的典型方案是：

- EF Core 的上下文和实体类位于 .NET Core 类库中。
- .NET Core 控制台应用程序或 web 应用程序引用类库。

还可以[将迁移代码放在与 EF Core 上下文分离的类库中](#)。

其他目标框架

包管理器控制台工具适用于 .NET Core 或 .NET Framework 项目。.NET Standard 类库中具有 EF Core 模型的应用可能没有 .NET Core 或 .NET Framework 项目。例如，这适用于 Xamarin 和通用 Windows 平台应用。在这种情况下，你可以创建一个 .NET Core 或 .NET Framework 的控制台应用程序项目，该项目的唯一用途是充当工具的启动项目。项目可以是不包含实际代码 — 的虚拟项目，只需为工具提供目标。

为什么需要虚拟项目？如前文所述，这些工具必须在设计时执行应用程序代码。为此，需要使用 .NET Core 或 .NET Framework 运行时。当 EF Core 模型位于面向 .NET Core 或 .NET Framework 的项目中时，EF Core 工具会借用项目中的运行时。如果 EF Core 模型在 .NET Standard 类库中，则无法执行此操作。.NET Standard 不是实际的 .NET 实现；它是 .NET 实现所必须支持的一组 API 的规范。因此 .NET Standard 不足以执行应用程序代码 EF Core 工具。你创建的用于启动项目的虚拟项目提供了一个具体的目标平台，工具可在其中加载 .NET Standard 类库。

ASP.NET Core 环境

若要为 ASP.NET Core 项目指定环境，请在运行命令之前设置 env: ASPNETCORE_ENVIRONMENT 。

通用参数

下表显示了所有 EF Core 命令共有的参数：

参数	说明
-Context <字符串>	要使用的 <code>DbContext</code> 类。仅命名空间或完全限定类名。如果省略此参数，EF Core 将查找上下文类。如果有多个上下文类，则此参数是必需的。
-Project <字符串>	目标项目。如果省略此参数，则 <code>dotnet ef</code> 的 <code>dotnet</code> 将用作目标项目。
-StartupProject <字符串>	启动项目。如果省略此参数，则使用 <code>dotnet ef</code> 中的 <code>dotnet</code> 作为目标项目。
-Verbose	显示详细输出。

若要显示有关命令的帮助信息，请使用 PowerShell `Get-Help` 命令。

TIP

Context、Project 和 StartupProject 参数支持 tab 补全。

Add-Migration

添加新的迁移。

参数：

参数	说明
-Name <String>	迁移的名称。这是一个位置参数，并且是必需的。
-OutputDir <字符串>	要使用的目录（和子命名空间）。路径相对于目标项目目录。默认值为 "迁移"。

Drop 数据库

删除数据库。

参数：

-WhatIf	显示要删除的数据库，但不删除它。
----------------	------------------

Get-DbContext

获取有关 `DbContext` 类型的信息。

Remove-Migration

删除上一次迁移(回滚针对迁移进行的代码更改)。

参数：

-Force	恢复迁移(回滚应用于数据库的更改)。
---------------	--------------------

Scaffold-DbContext

为数据库的 `DbContext` 和实体类型生成代码。为了使 `Scaffold-DbContext` 生成实体类型，数据库表必须具有主键。

参数：

-连接 <字符串 >	用于连接到数据库的连接字符串。对于 ASP.NET Core 2.x 项目，值可以是名称 =<连接字符串 > 的名称。在这种情况下，该名称来自为项目设置的配置源。这是一个位置参数，并且是必需的。
-Provider <字符串 >	要使用的提供程序。通常，这是 NuGet 包的名称，例如： <code>Microsoft.EntityFrameworkCore.SqlServer</code> 。这是一个位置参数，并且是必需的。
-OutputDir <字符串 >	要在其中放置文件的目录。路径相对于项目目录。
-ContextDir <字符串 >	要在其中放置 <code>DbContext</code> 文件的目录。路径相对于项目目录。
-Context <字符串 >	要生成的 <code>DbContext</code> 类的名称。
-架构 <String [] >	要为其生成实体类型的表的架构。如果省略此参数，则包括所有架构。
-表 <String [] >	要为其生成实体类型的表。如果省略此参数，则包括所有表。
-DataAnnotations	使用属性配置模型(如果可能)。如果省略此参数，则只使用 Fluent API。
-UseDatabaseNames	使用表和列的名称与数据库中显示的名称完全相同。如果省略此参数，则更改数据库名称以更严格地 C# 符合名称样式约定。

II	II
-Force	覆盖现有文件。

示例：

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

示例：仅基架选定的表，并在具有指定名称的单独文件夹中创建上下文：

```
Scaffold-DbContext "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables "Blog", "Post" -ContextDir Context -  
Context BlogContext
```

Script-Migration

生成一个 SQL 脚本，该脚本将所选迁移中的所有更改应用于另一个选定的迁移。

参数：

II	II
-来自<字符串>	开始迁移。可以按名称或 ID 识别迁移。数字0是一个特殊情况，表示在第一次迁移之前。默认值为 0。
-要<字符串>	结束迁移。默认为上次迁移。
-幂等	生成可用于任何迁移的数据库的脚本。
-输出 <字符串>	要向其写入结果的文件。如果省略此参数，则会在创建应用的运行时文件所在的同一文件夹中创建具有生成名称的文件，例如：/obj/Debug/netcoreapp2.1/ghbkztf.sql/。

TIP

To、From 和 Output 参数支持tab补全。

以下示例使用迁移名称创建用于 InitialCreate 迁移的脚本。

```
Script-Migration -To InitialCreate
```

以下示例使用迁移 ID，为 InitialCreate 迁移后的所有迁移创建一个脚本。

```
Script-Migration -From 20180904195021_InitialCreate
```

Update-Database

将数据库更新到上次迁移或指定迁移。

-迁移<字符串 >	目标迁移。可以按名称或 ID 识别迁移。数字0是一种特殊情况，表示在第一次迁移之前，并导致还原所有迁移。如果未指定迁移，则该命令默认为上一次迁移。

TIP

Migration 参数支持tab补全。

下面的示例将还原所有迁移。

```
Update-Database -Migration 0
```

下面的示例将数据库更新为指定的迁移。第一个使用迁移名称，第二个使用迁移 ID：

```
Update-Database -Migration InitialCreate  
Update-Database -Migration 20180904195021_InitialCreate
```

其他资源

- [迁移](#)
- [反向工程](#)

Entity Framework Core 工具参考-.NET CLI

2020/3/11 •

用于 Entity Framework Core 的命令行接口(CLI)工具执行设计时开发任务。例如，他们基于现有数据库创建[迁移](#)、[应用迁移](#)和生成模型的代码。命令是跨平台[dotnet](#)命令的扩展，它是[.NET Core SDK](#)的一部分。这些工具适用于 .NET Core 项目。

如果使用的是 Visual Studio，我们建议改用[包管理器控制台工具](#)：

- 它们自动使用在包管理器控制台中选择的当前项目，而无需手动切换目录。
- 在命令完成后，它们会自动打开由命令生成的文件。

安装工具

安装过程取决于项目类型和版本：

- EF Core 1.x
- ASP.NET Core 版本2.1 及更高版本
- EF Core 2.x
- EF Core 1.x

EF Core 1.x

- `dotnet ef` 必须安装为全局或本地工具。大多数开发人员会使用以下命令将 `dotnet ef` 安装为全局工具：

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

你还可以使用 `dotnet ef` 作为本地工具。若要将其用作本地工具，请使用[工具清单文件](#)还原项目的依赖项，将该项目声明为[工具依赖项](#)。

- 安装[.NET Core SDK](#)。
- 安装最新的 `Microsoft.EntityFrameworkCore.Design` 包。

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

ASP.NET Core 2.1 +

- 安装当前[.NET Core SDK](#)。即使有 Visual Studio 2017 的最新版本，也必须安装 SDK。

这是 ASP.NET Core 2.1 + 所需的全部，因为 `Microsoft.EntityFrameworkCore.Design` 包包含在[AspNetCore 元包](#)中。

EF Core 2.x (不 ASP.NET Core)

`dotnet ef` 命令包含在 .NET Core SDK 中，但要启用这些命令，必须安装 `Microsoft.EntityFrameworkCore.Design` 包。

- 安装当前[.NET Core SDK](#)。即使使用最新版本的 Visual Studio，也必须安装 SDK。
- 安装最新的稳定 `Microsoft.EntityFrameworkCore.Design` 包。

```
dotnet add package Microsoft.EntityFrameworkCore.Design
```

EF Core 1.x

- 安装 .NET Core SDK 版本 2.1.200。更高版本与用于 EF Core 1.0 和 1.1 的 CLI 工具不兼容。
- 通过修改应用程序的 `global.asax` 文件，将该应用程序配置为使用 2.1.200 SDK 版本。此文件通常包含在解决方案目录中（项目上面的一个）。
- 编辑项目文件，并将 `Microsoft.EntityFrameworkCore.Tools.DotNet` 作为 `DotNetCliToolReference` 项添加。指定最新的 1.x 版本，例如：1.1.6。请参阅本部分末尾的项目文件示例。
- 安装最新版本的 `Microsoft.EntityFrameworkCore.Design` 包，例如：

```
dotnet add package Microsoft.EntityFrameworkCore.Design -v 1.1.6
```

添加这两个包引用后，项目文件将如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="1.1.6"
      PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
      Version="1.1.6" />
  </ItemGroup>
</Project>
```

不向引用此项目的项目公开带有 `PrivateAssets="All"` 的包引用。对于通常只在开发过程中使用的包，此限制特别有用。

验证安装

运行以下命令以验证是否正确安装了 EF Core CLI 工具：

```
dotnet restore
dotnet ef
```

命令的输出标识所使用的工具的版本：

```

 _/\_
----/   \\
|_.  \|\\
|__||_| |_) \\\
|_| ||_| \/_| //\\\
|__||_|     /  \\\\
                           Entity Framework Core .NET Command-line Tools 2.1.3-rtm-32065
                           <Usage documentation follows, not shown.>
```

使用工具

在使用这些工具之前，您可能需要创建一个启动项目或设置环境。

目标项目和启动项目

命令引用 [项目](#) 和 [启动项目](#)。

- 该项目也称为 [目标项目](#)，因为它是命令在其中添加或删除文件的位置。默认情况下，当前目录中的项目是目标项目。您可以使用 `--project` 选项指定其他项目作为目标项目。
- [启动项目](#) 是工具生成和运行的项目。这些工具必须在设计时执行应用程序代码，以获取有关项目的信息，例如数据库连接字符串和模型的配置。默认情况下，当前目录中的项目是启动项目。您可以使用 "`--startup-project`" 选项指定一个不同的项目作为启动项目。

启动项目和目标项目通常是同一个项目。它们是不同的项目的典型方案是：

- EF Core 的上下文和实体类位于 .NET Core 类库中。
- .NET Core 控制台应用程序或 web 应用程序引用类库。

还可以[将迁移代码放在与 EF Core 上下文分离的类库中](#)。

其他目标框架

CLI 工具适用于 .NET Core 项目和 .NET Framework 项目。.NET Standard 类库中具有 EF Core 模型的应用可能没有 .NET Core 或 .NET Framework 项目。例如，这适用于 Xamarin 和通用 Windows 平台应用。在这种情况下，你可以创建一个仅供其使用的 .NET Core 控制台应用项目，作为工具的启动项目。项目可以是不包含实际代码的虚拟项目，只需为工具提供目标。

为什么需要虚拟项目？如前文所述，这些工具必须在设计时执行应用程序代码。为此，需要使用 .NET Core 运行时。当 EF Core 模型位于面向 .NET Core 或 .NET Framework 的项目中时，EF Core 工具会借用项目中的运行时。如果 EF Core 模型在 .NET Standard 类库中，则无法执行此操作。.NET Standard 不是实际的 .NET 实现；它是 .NET 实现所必须支持的一组 API 的规范。因此 .NET Standard 不足以执行应用程序代码 EF Core 工具。你创建的用于启动项目的虚拟项目提供了一个具体的目标平台，工具可在其中加载 .NET Standard 类库。

ASP.NET Core 环境

若要为 ASP.NET Core 项目指定环境，请在运行命令之前设置 `ASPNETCORE_ENVIRONMENT` 环境变量。

常用选项

	示例	说明
	<code>--json</code>	显示 JSON 输出。
<code>-c</code>	<code>--context <DBCONTEXT></code>	要使用的 <code>DbContext</code> 类。仅命名空间或完全限定类名。如果省略此选项，EF Core 将查找上下文类。如果有多个上下文类，则需要此选项。
<code>-p</code>	<code>--project <PROJECT></code>	目标项目的项目文件夹的相对路径。默认值为当前文件夹。
<code>-s</code>	<code>--startup-project <PROJECT></code>	启动项目的项目文件夹的相对路径。默认值为当前文件夹。
	<code>--framework <FRAMEWORK></code>	目标框架 的 目标框架名字对象 。当项目文件指定多个目标框架，并想要选择其中一个时，请使用。

	--configuration <CONFIGURATION>	生成配置，例如：Debug 或 Release。
	--runtime <IDENTIFIER>	要为其还原包的目标运行时的标识符。有关运行时标识符 (RID) 的列表，请参阅 RID 目录 。
-h	--help	显示帮助信息。
-v	--verbose	显示详细输出。
	--no-color	不要为输出着色。
	--prefix-output	用 level 作为输出前缀。

dotnet ef 数据库删除

删除数据库。

选项：

	--	--
-f	--force	不要确认。
	--dry-run	显示要删除的数据库，但不删除它。

dotnet ef 数据库更新

将数据库更新到上次迁移或指定迁移。

参数：

ARGUMENT	--
<MIGRATION>	目标迁移。可以按名称或 ID 识别迁移。数字0是一种特殊情况，表示在第一次迁移之前，并导致还原所有迁移。如果未指定迁移，则该命令默认为上一次迁移。

下面的示例将数据库更新为指定的迁移。第一个使用迁移名称，第二个使用迁移 ID：

```
dotnet ef database update InitialCreate
dotnet ef database update 20180904195021_InitialCreate
```

dotnet ef dbcontext 信息

获取有关 `DbContext` 类型的信息。

dotnet ef dbcontext 列表

列出可用 `DbContext` 类型。

dotnet ef dbcontext 基架

为数据库的 `DbContext` 和实体类型生成代码。为了使此命令生成实体类型，数据库表必须具有主键。

参数：

ARGUMENT	DE
<code><CONNECTION></code>	用于连接到数据库的连接字符串。对于 ASP.NET Core 2.x 项目，值可以是名称 = <code><连接字符串></code> 的名称。在这种情况下，该名称来自为项目设置的配置源。
<code><PROVIDER></code>	要使用的提供程序。通常，这是 NuGet 包的名称，例如： <code>Microsoft.EntityFrameworkCore.SqlServer</code> 。

选项：

	DE	DE
<code>-d</code>	<code>--data-annotations</code>	使用属性配置模型(如果可能)。如果省略此选项，则只使用 Fluent API。
<code>-c</code>	<code>--context <NAME></code>	要生成的 <code>DbContext</code> 类的名称。
	<code>--context-dir <PATH></code>	要在其中放置 <code>DbContext</code> 类文件的目录。路径相对于项目目录。命名空间是从文件夹名称派生的。
<code>-f</code>	<code>--force</code>	覆盖现有文件。
<code>-o</code>	<code>--output-dir <PATH></code>	要在其中放置实体类文件的目录。路径相对于项目目录。
	<code>--schema <SCHEMA_NAME> ...</code>	要为其生成实体类型的表的架构。若要指定多个架构，请对每个架构重复 <code>--schema</code> 。如果省略此选项，则包括所有架构。
<code>-t</code>	<code>--table <TABLE_NAME> ...</code>	要为其生成实体类型的表。若要指定多个表，请对每个表重复 <code>-t</code> 或 <code>--table</code> 。如果省略此选项，则包括所有表。
	<code>--use-database-names</code>	使用表和列的名称与数据库中显示的名称完全相同。如果省略此选项，则更改数据库名称以更严格地 C# 符合名称样式约定。

下面的示例基架所有架构和表，并将新文件放在 `Models` 文件夹中。

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

以下示例仅基架选定的表，并在具有指定名称的单独文件夹中创建上下文：

```
dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -o Models -t Blog -t Post --context-dir Context -c BlogContext
```

dotnet ef 迁移添加

添加新的迁移。

参数:

ARGUMENT	DESCRIPTION
<NAME>	迁移的名称。

选项:

	DESCRIPTION	DESCRIPTION
-o	--output-dir <PATH>	要使用的目录(和子命名空间)。路径相对于项目目录。默认值为 "迁移"。

dotnet ef 迁移列表

列出可用迁移。

dotnet ef 迁移删除

删除上一次迁移(回滚针对迁移进行的代码更改)。

选项:

	DESCRIPTION	DESCRIPTION
-f	--force	恢复迁移(回滚应用于数据库的更改)。

dotnet ef 迁移脚本

从迁移生成 SQL 脚本。

参数:

ARGUMENT	DESCRIPTION
<FROM>	开始迁移。可以按名称或 ID 识别迁移。数字0是一个特殊情况，表示在第一次迁移之前。默认值为 0。
<TO>	结束迁移。默认为上次迁移。

选项:

	DESCRIPTION	DESCRIPTION
-o	--output <FILE>	要写入脚本的文件。

	<code>--i</code>	<code>--idempotent</code>	
			生成可用于任何迁移的数据库的脚本。

以下示例创建用于 InitialCreate 迁移的脚本：

```
dotnet ef migrations script 0 InitialCreate
```

以下示例在 InitialCreate 迁移之后为所有迁移创建一个脚本。

```
dotnet ef migrations script 20180904195021_InitialCreate
```

其他资源

- [迁移](#)
- [反向工程](#)

设计时 DbContext 创建

2020/3/11 ·

某些 EF Core 工具命令(例如, [迁移命令](#))需要在设计时创建一个派生的 `DbContext` 实例, 以便收集有关应用程序的实体类型及其如何映射到数据库架构的详细信息。在大多数情况下, 希望创建的 `DbContext` 的配置方式类似于在[运行时](#)对其进行配置的方式。

工具可通过多种方式来创建 `DbContext` :

从应用程序服务

如果启动项目使用[ASP.NET Core Web 主机](#)或[.Net Core 泛型主机](#), 则这些工具将尝试从应用程序的服务提供程序获取 `DbContext` 对象。

工具首先尝试通过调用 `Program.CreateHostBuilder()`, 调用 `Build()`, 然后访问 `Services` 属性来获取服务提供程序。

```
public class Program
{
    public static void Main(string[] args)
        => CreateHostBuilder(args).Build().Run();

    // EF Core uses this method at design time to access the DbContext
    public static IHostBuilder CreateHostBuilder(string[] args)
        => Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(
                webBuilder => webBuilder.UseStartup<Startup>());
}

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
        => services.AddDbContext<ApplicationDbContext>();

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseHttpsRedirection();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }
}
```

NOTE

创建新的 ASP.NET Core 应用程序时, 默认情况下会包括此挂接。

在应用程序的服务提供程序中, 需要将 `DbContext` 本身及其构造函数中的任何依赖项注册为服务。为此, 可以在将 `DbContextOptions<TContext>` 实例作为参数并使用 `AddDbContext<TContext>` 方法的 `DbContext` 上使用构造函数。

使用不带参数的构造函数

如果无法从应用程序服务提供程序获得 `DbContext`, 则工具会在项目中查找派生 `DbContext` 类型。然后, 它们尝试使用不带参数的构造函数创建实例。如果 `DbContext` 是使用 [OnConfiguring](#) 方法配置的, 则这可能是默认构造函

数。

从设计时工厂

还可以通过实现 `IDesignTimeDbContextFactory<TContext>` 接口告诉工具如何创建 `DbContext`: 如果实现此接口的类在与派生 `DbContext` 相同的项目中或在应用程序的启动项目中找到, 则这些工具将绕过其他创建 `DbContext` 的方法, 而改用设计时工厂。

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Infrastructure;

namespace MyProject
{
    public class BloggingContextFactory : IDesignTimeDbContextFactory<BloggingContext>
    {
        public BloggingContext CreateDbContext(string[] args)
        {
            var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
            optionsBuilder.UseSqlite("Data Source=blog.db");

            return new BloggingContext(optionsBuilder.Options);
        }
    }
}
```

NOTE

`args` 参数当前未使用。跟踪从工具中指定设计时参数的功能时出现[问题](#)。

如果需要在设计时与运行时不同的情况下配置 `DbContext`, 则设计时工厂特别有用。如果 `DbContext` 构造函数采用其他参数, 但未在 DI 中注册, 则如果不使用 DI, 或者出于某种原因而不希望在 ASP.NET Core 应用程序的 `Main` 类中使用 `BuildWebHost` 方法。

设计时服务

2020/3/11 •

这些工具使用的某些服务仅在设计时使用。这些服务独立于 EF Core 的运行时服务进行管理，以防止这些服务与你的应用程序一起部署。若要替代其中的某个服务（例如，生成迁移文件的服务），请将 `IDesignTimeServices` 的实现添加到启动项目。

```
class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
        => services.AddSingleton<IMigrationsCodeGenerator, MyMigrationsCodeGenerator>();
}
```

Entity Framework 6

2020/4/8 • [Edit Online](#)

实体框架 6 (EF6) 是经试验和测试的关系映射器 (O/RM)，适用于 .NET 的对象，其功能和稳定性经过了多年的开发和调试。

作为 O/RM，EF6 降低了关系方面和面向对象的方面之间的阻抗不匹配，使开发人员能够使用表示应用程序域的强类型 .NET 对象来编写应用程序，该应用程序可与存储在关系数据库中的数据交互，同时使开发人员无需再编写大部分的数据访问“管道”代码。

EF6 可实现许多热门 O/RM 功能：

- 不依赖于任何 EF 类型的 [POCO](#) 实体类的映射
- 自动更改跟踪
- 标识解析和工作单元
- 预先、延迟和显式加载
- 使用 [LINQ\(语言集成查询\)](#) 转换强类型查询
- 丰富的映射功能，可支持：
 - 一对一、一对多和多对多关系
 - 继承(每个层次结构一张表、每个类型一张表和每个具体类一张表)
 - 复杂类型
 - 存储过程
- 通过可视化设计器创建实体模型。
- 通过编写代码创建实体模型的“Code First”体验。
- 既可从现有数据库生成模型，然后手动编辑，也可从头开始创建模型，然后用于生成新的数据库。
- 与 .NET Framework 应用程序模型(包括 ASP.NET)集成，并通过数据绑定与 WPF 和 WinForms 集成。
- 基于 ADO.NET 的数据库连接和可用于连接到 SQL Server、Oracle、MySQL、SQLite、PostgreSQL、DB2 等的众多[提供程序](#)。

应使用 EF6 还是 EF Core？

EF Core 是更现代、可扩展的轻量级实体框架版本，与 EF6 的功能和优点非常相似。EF Core 则完全进行了重写，包含许多 EF6 没有的新功能，但还是缺少 EF6 中最高级的一些映射功能。如果功能集与需求匹配，请考虑在新应用程序中使用 EF Core。[比较 EF Core 和 EF6](#) 中更详细地讨论了此选项。

入门

将 EntityFramework NuGet 包添加到项目或安装[适用于 Visual Studio 的 Entity Framework Tools](#)。然后观看视频、阅读教程和高级文档，以充分利用 EF6。

过去的实体框架版本

本文档针对的是最新版本的实体框架 6，但其中大部分内容也适用于过去的版本。请查看[新增功能](#)和[过去的版本](#)，了解 EF 版本和其中引入的功能的完整列表。

EF6 中的新增功能

2020/4/8 ·

强烈建议使用最新发布的实体框架版本，以确保获得最新功能和最高稳定性。但我们也意识到用户可能需要使用以前的版本，或想要尝试最新预发行版中的新改进。若要安装特定版本的 EF，请参阅[获取实体框架](#)。

EF 6.4.0

EF 6.4.0 运行时已于 2019 年 12 月发布到 NuGet。EF 6.4 的主要目标是优化 EF 6.3 中提供的功能和方案。请参阅[Github 上的重要修复列表](#)。

EF 6.3.0

EF 6.3.0 运行时已于 2019 年 9 月发布到 NuGet。此版本的主要目标是帮助将使用 EF 6 的现有应用程序迁移到 .NET Core 3.0。社区还提供了多个 bug 修复和增强功能。有关详细信息，请参阅每个 6.3.0 [里程碑](#) 中关闭的问题。下面是一些更值得注意的事项：

- 支持 .NET Core 3.0
 - 除了 .NET Framework 4.x, EntityFramework 包现在还面向 .NET Standard 2.1。
 - 这意味着 EF 6.3 是跨平台的，并在 Windows 之外的其他操作系统（如 Linux 和 macOS）上受支持。
 - 已重新编写迁移命令以在进程外执行，并使用 SDK 样式的项目。
- 支持 SQL Server HierarchyId。
- 提高了与 Roslyn 和 NuGet PackageReference 的兼容性。
- 添加了 `ef6.exe` 实体工具，用于启用、添加、编写脚本和应用程序集中的迁移。这会替换 `migrate.exe`。

EF 设计器支持

目前不支持直接在 .NET Core 或 .NET Standard 项目上或在 SDK 样式的 .NET Framework 项目上使用 EF 设计器。

可以通过在同一解决方案中将 EDMX 文件以及为实体和 DbContext 生成的类作为链接文件添加到 .NET Core 3.0 或 .NET Standard 2.1 项目中来解决此限制。

在项目文件中，链接文件将如下所示：

```
<ItemGroup>
  <EntityDeploy Include="..\EdmxDesignHost\Entities.edmx" Link="Model\Entities.edmx" />
  <Compile Include="..\EdmxDesignHost\Entities.Context.cs" Link="Model\Entities.Context.cs" />
  <Compile Include="..\EdmxDesignHost\Thing.cs" Link="Model\Thing.cs" />
  <Compile Include="..\EdmxDesignHost\Person.cs" Link="Model\Person.cs" />
</ItemGroup>
```

请注意，EDMX 文件与 EntityDeploy 生成操作关联。这是一个特殊的 MSBuild 任务（现已包含在 EF 6.3 包中），它负责将 EF 模型作为嵌入式资源添加到目标程序集中（或将其复制为输出文件夹中的文件，具体取决于 EDMX 中的“元数据项目处理”设置）。有关如何进行此设置的详细信息，请参阅[EDMX .NET Core 示例](#)。

警告：请确保定义“实际”.edmx 文件的旧样式（而非 SDK 样式）.NET Framework 项目在定义 .sln 文件内的链接的项目之前。否则，当在设计器中打开 .edmx 文件时，将看到错误消息“该实体框架在当前为目标框架不可用。可以更改项目的目标框架或在 XmlEditor 中编辑模型”。

过去的版本

[过去的版本](#) 页包含过去所有版本的 EF 的存档以及各版本中引入的主要功能。

实体框架的过去版本

2020/3/18 •

第一版实体框架在2008中发布，作为 .NET Framework 3.5 SP1 和 Visual Studio 2008 SP1 的一部分。

从 EF 4.1 版本开始，该版本已作为[EntityFramework NuGet 包](#)提供-当前 NuGet.org 上最热门的包之一。

在版本4.1 和5.0 之间，EntityFramework NuGet 包扩展了作为 .NET Framework 一部分提供的 EF 库。

从版本6开始，EF 变成了一个开源项目，同时从 .NET Framework 完全移入。这意味着，当你将 EntityFramework 版本 6 NuGet 包添加到应用程序时，你将获得 EF 库的完整副本，该副本不依赖于 .NET Framework 附带的 EF 位。这有助于在一定程度上加快新功能的开发和交付速度。

2016年6月发布 EF Core 1.0。EF Core 基于新的代码库，旨在作为 EF 的更轻型且可扩展的版本。目前 EF Core 是 Microsoft 的实体框架团队开发的主要焦点。这意味着没有为 EF6 计划的新的主要功能。但是，EF6 仍将保持为开源项目和受支持的 Microsoft 产品。

下面是过去版本的列表，采用反向时间顺序，其中包含有关每个版本中引入的新功能的信息。

Visual Studio 2017 15.7 中的 EF 工具更新

2018 年 5 月，我们在 Visual Studio 2017 15.7 中发布了更新后的 EF 工具。该版本包括对一些常见点的改进：

- 修复了多个用户界面辅助功能 bug
- 从现有数据库生成模型时 SQL Server 性能退化的解决方法 [#4](#)
- 支持适用于 SQL Server 上较大模型的更新模型 [#185](#)

此新版本的 EF 工具中的另一项改进是，在新项目中创建模型时会安装 EF 6.2 运行时。借助较旧版本的 Visual Studio，可通过安装相应版本的 NuGet 包来使用 EF 6.2 运行时（以及以前任何版本的 EF）。

EF 6.2。0

EF 6.2 运行时已于 2017 年 10 月发布到 NuGet。在我们开源社区参与者的努力下，EF 6.2 包括大量的 [bug 修复](#) 和 [产品增强功能](#)。

下表简要列出了影响 EF 6.2 运行时的最重要的更改：

- 通过加载持久性缓存中已完成的 Code First 模型来加快启动 [#275](#)
- 采用 Fluent API 定义索引 [#274](#)
- 通过 DbFunctions.Like() 编写在 SQL 中转换为 LIKE 的 LINQ 查询 [#241](#)
- Migrate.exe 现支持脚本选项 [#240](#)
- EF6 现在可使用 SQL Server 中的序列生成的键值 [#165](#)
- 更新 SQL Azure 执行策略的暂时性错误列表 [#83](#)
- Bug: 重试查询或 SQL 命令失败，“另一 SqlParameterCollection 中已包含 SqlParameter”[#81](#)
- Bug:DbQuery.ToString() 评估在调试程序中经常超时 [#73](#)

EF EF6.1。3

EF ef6.1.3 运行时发布到了2015年10月的 NuGet。此版本仅包含对6.1.2 版本报告的高优先级缺陷和回归的修补程序。修复包括：

- 查询：EF 6.1.2 中的回归：OUTER 适用于1:1 关系和 "let" 子句的已引入和更复杂的查询

- 继承类中隐藏基类属性的 TPT 问题
- 当文本中包含单词 "DbMigration" 时, Sql 失败
- 创建供 unionall 和交集平展支持的兼容性标志
- 具有多个包含的查询在6.1.2 中不起作用(在6.1.1 中工作)
- 从 EF 6.1.1 升级到6.1.2 后, 出现 SQL 语法错误

EF 6.1。2

EF 6.1.2 运行时已发布到2014年12月的 NuGet。此版本主要涉及 bug 修复。我们还接受了社区成员的几个值得注意的更改:

- 可以从应用程序/web.config 文件配置查询缓存参数

```
<entityFramework>
  <queryCache size='1000' cleaningIntervalInSeconds='-1' />
</entityFramework>
```

- DbMigration 上的 SqlFile 和 SqlResource 方法允许您运行存储为文件或嵌入资源的 SQL 脚本。

EF 6.1。1

EF 6.1.1 运行时已发布到2014年6月。此版本包含许多人遇到的问题的修补程序。其他:

- 设计器:在 EF6 设计器中打开具有十进制精度的 EF5 edmx 时出错
- LocalDB 的默认实例检测逻辑不适用于 SQL Server 2014

EF 6.1。0

EF 6.1.0 运行时发布到了2014年3月的 NuGet。此次要更新包含大量新功能:

- **工具合并**为创建新的 EF 模型提供了一种一致的方法。此功能[扩展了 ADO.NET 实体数据模型向导以支持创建 Code First 模型](#), 包括从现有数据库进行反向工程。这些功能之前已在 EF Power Tools 中提供 Beta 版质量。
- **处理事务提交失败**会提供 CommitFailureHandler, 它利用新引入的截取事务操作的能力。CommitFailureHandler 允许在提交事务的同时从连接故障中自动恢复。
- **IndexAttribute**允许通过在 Code First 模型中的属性(或属性)上放置 `[Index]` 特性来指定索引。然后 Code First 将在数据库中创建相应的索引。
- **公共映射 API**提供对信息 EF 的访问, 以了解如何将属性和类型映射到数据库中的列和表。在以前的版本中, 此 API 是内部的。
- **通过 App/web.config 文件配置侦听器的能力**允许添加侦听器, 而无需重新编译应用程序。
- **DatabaseLogger**是一个新的侦听器, 可让你轻松地将所有数据库操作记录到文件中。与上一项功能结合使用, 可以轻松地针对已[部署的应用程序的数据库操作进行日志记录](#), 而无需重新编译。
- 改进了**迁移模型更改检测**, 使基架迁移更准确;还增强了更改检测过程的性能。
- **性能改进**, 包括在初始化期间降低数据库操作、在更多方案中优化 null 相等性比较、更快速地生成视图(创建模型)以及更有效地具体化具有多个关联的跟踪实体。

EF 6.0。2

EF 6.0.2 运行时已发布到2013年12月的 NuGet。此修补程序版本仅限于修复在 EF6 版本中引入的问题(自 EF5 以来性能/行为的回归)。

EF 6.0。1

EF 6.0.1 运行时已发布到2013年10月的 NuGet, 因为后者嵌入到了几个月之前的 Visual Studio 版本中。此修补程

序版本仅限于修复在 EF6 版本中引入的问题(自 EF5 以来性能/行为的回归)。最值得注意的更改是解决 EF 模型的预热过程中的某些性能问题。这一点非常重要, 因为预热性能是 EF6 的一个重点领域, 而这些问题取消 EF6 的一些其他性能。

EF 6.0

EF 6.0 运行时发布到了2013年10月的 NuGet。这是[EntityFramework NuGet 包](#)中的第一个版本, 在该版本中, 不依赖于 .NET Framework 中的 EF 位。将运行时的剩余部分移到 NuGet 包需要对现有代码进行大量的重大更改。有关升级所需的手动步骤的详细信息, 请参阅[升级到实体框架 6 部分](#)。

此版本包含许多新功能。以下功能适用于使用 Code First 或 EF 设计器创建的模型:

- [异步查询和保存](#) 增加了对 .net 4.5 中引入的基于任务的异步模式的支持。
- [连接复原](#) 允许从暂时性连接故障中自动恢复。
- [基于代码的配置](#) 使你能够在代码中执行配置(传统上在配置文件中执行)。
- [依赖项解析](#) 引入了对服务定位器模式的支持, 并分解了某些功能, 这些功能可以替换为自定义实现。
- [拦截/SQL 日志记录](#) 提供低级别的构建基块, 用于截获 EF 操作, 同时构建简单的 SQL 日志记录。
- [使用模拟 framework 或编写你自己的测试](#) 时, 可测试性改进可以更轻松地为 DbContext 和 DbSet 创建测试双精度。
- [现在可以使用已打开的 DbConnection 创建 DbContext](#), 这会启用在创建上下文时可以打开连接的情况(例如, 在不能保证连接状态的组件之间共享连接)的情况。
- [改进的事务支持](#) 为框架的外部事务提供支持, 并改进了在框架中创建事务的方式。
- [.Net 4.0 上的枚举、空间和更好的性能](#)-通过将已在 .NET Framework 中的核心组件移到 EF NuGet 包中, 我们现在可以提供枚举支持、空间数据类型和 .net 4.0 上 EF5 的性能改进。
- [可枚举的性能得到改进。包含在 LINQ 查询中。](#)
- 缩短了预热时间(视图生成), 尤其是对于大型模型。
- [可插接式复数形式 & Singularization 服务](#)。
- 现在支持实体类上 Equals 或 GetHashCode 的自定义实现。
- [DbSet.AddRange/RemoveRange](#) 提供一种优化的方式来添加或删除集中的多个实体。
- [DbChangeTracker HasChanges](#) 提供一种简单有效的方法来查看是否有任何挂起的更改保存到数据库中。
- [SqlCeFunctions](#) 提供与 SqlFunctions 等效的 SQL Compact。

以下功能仅适用于 Code First:

- [自定义 Code First 约定](#) 允许编写自己的约定, 以帮助避免重复配置。我们提供了一个简单的轻型约定 API 以及一些更复杂的构建基块, 使你能够创作更复杂的约定。
- 现在支持 [Code First 映射到插入/更新/删除存储过程](#)。
- [幂等迁移脚本](#) 允许生成一个 SQL 脚本, 该脚本可以将任何版本的数据库升级到最新版本。
- 可 [配置的迁移历史记录表](#) 允许自定义迁移历史记录表的定义。这对于需要适当的数据类型等的数据库提供程序特别有用, 因为需要为迁移历史记录表指定这些数据类型才能正常工作。
- 当使用迁移时, 或 Code First 自动为您创建数据库时, [每个数据库的多个上下文将删除每个数据库的一个 Code First 模型的以前限制](#)。
- [DbModelBuilder](#) 是一个新的 Code First API, 它允许在一个位置配置 Code First 模型的默认数据库架构。以前 Code First 的默认架构已硬编码为 "dbo", 以及通过 ToTable API 配置表所属架构的唯一方法。
- 在将配置类与 Code First 熟知 API 一起使用时, [AddFromAssembly](#) 方法可让你轻松地添加在程序集中定义的所有配置类。
- [自定义迁移操作](#) 使您能够添加要在基于代码的迁移中使用的其他操作。
- 对于使用 Code First 创建的数据库, 默认事务隔离级别将更改为 READ_COMMITTED_SNAPSHOT, 从而实现更高的可伸缩性和更少的死锁。
- [实体和复杂类型现在可以是 nestedinside 类](#)。

EF 5.0

2012年8月发布了 EF 5.0.0 运行时。此版本引入了一些新功能，包括枚举支持、表值函数、空间数据类型和各种性能改进。

Visual Studio 2012 中的 Entity Framework Designer 还引入了对每个模型的多个关系图的支持、设计图面上的形状的颜色以及存储过程的批处理导入。

下面是专门为 EF 5 版本组合在一起的内容列表：

- [EF 5 Release Post](#)
- EF5 中的新增功能
 - [Code First 中的枚举支持](#)
 - [EF 设计器中的枚举支持](#)
 - [Code First 中的空间数据类型](#)
 - [EF 设计器中的空间数据类型](#)
 - [提供程序对空间类型的支持](#)
 - [表值函数](#)
 - [每个模型多个关系图](#)
- 设置模型
 - [创建模型](#)
 - [连接和模型](#)
 - [性能注意事项](#)
 - [使用 Microsoft SQL Azure](#)
 - [配置文件设置](#)
 - [术语表](#)
 - Code First
 - [Code First 到新数据库\(演练和视频\)](#)
 - [Code First 现有数据库\(演练和视频\)](#)
 - [约定](#)
 - [数据注释](#)
 - [熟知 API-配置/映射属性 & 类型](#)
 - [熟知 API 配置关系](#)
 - [通过 VB.NET 的流畅 API](#)
 - [Code First 迁移](#)
 - [自动 Code First 迁移](#)
 - [Debug.exe](#)
 - [定义 Dbset](#)
 - EF 设计器
 - [Model First \(演练和视频\)](#)
 - [Database First \(演练和视频\)](#)
 - [复杂类型](#)
 - [关联/关系](#)
 - [TPT 继承模式](#)
 - [TPH 继承模式](#)
 - [带有存储过程的查询](#)
 - [包含多个结果集的存储过程](#)
 - [插入、更新 & 删除与存储过程](#)
 - [将实体映射到多个表\(实体拆分\)](#)

- 将多个实体映射到一个表(表拆分)
- 定义查询
- 代码生成模板
- 恢复到 ObjectContext
- 使用模型
 - 使用 DbContext
 - 查询/查找实体
 - 使用关系
 - 正在加载相关实体
 - 使用本地数据
 - N 层应用程序
 - 原始 SQL 查询
 - 开放式并发模式
 - 使用代理
 - 自动检测更改
 - 无跟踪查询
 - 加载方法
 - 添加/附加和实体状态
 - 使用属性值
 - 与 WPF 的数据绑定(Windows Presentation Foundation)
 - 用 WinForms (Windows 窗体) 进行数据绑定

EF 4.3。1

Ef 4.3.1 运行时在 EF 4.3.0 之后不久发布到 2012 NuGet。此修补版本包含对 EF 4.3 版本的一些 bug 修复，并为使用 EF 4.3 与 Visual Studio 2012 的客户引入更好的 LocalDB 支持。

下面是专门为 EF 4.3.1 版本组合在一起的内容列表，为 EF 4.1 提供的大多数内容也适用于 EF 4.3：

- [EF 4.3.1 Release 博客文章](#)

EF 4。3

EF 4.3.0 运行时已发布到2012年2月的 NuGet。此版本包含新的 Code First 迁移功能，该功能允许增量更改 Code First 创建的数据库，因为 Code First 模型演变。

下面是专为 EF 4.3 版本组合在一起的内容列表，为 EF 4.1 提供的大多数内容也适用于 EF 4.3：

- [EF 4.3 发布后](#)
- [EF 4.3 基于代码的迁移演练](#)
- [EF 4.3 自动迁移演练](#)

EF 4。2

EF 4.2.0 运行时已发布到2011年11月的 NuGet。此版本包含对 EF 4.1.1 版本的 bug 修复。由于此发行版只包含 bug 修复，因此，它可能已成为 EF 4.1.2 修补程序版本，但我们选择迁移到4.2，以允许我们从在4.1 版本中使用的基于日期的修补程序版本号开始，并采用[语义 Versionsing](#)标准进行语义版本控制。

下面是专为 EF 4.2 版本组合在一起的内容列表，为 EF 4.1 提供的内容也适用于 EF 4.2：

- [EF 4.2 发布后](#)
- [Code First 演练](#)

- [Database First 演练 & 模型](#)

EF 4.1。1

EF 4.1.10715 运行时已发布到2011年7月的 NuGet。除了 bug 修复外，此修补程序版本还引入了一些组件，使设计时工具可以更轻松地处理 Code First 模型。这些组件由 Code First 迁移(包括在 EF 4.3 中)和 EF Power Tools 使用。

你会注意到，包的异常版本号4.1.10715。我们在决定采用[语义版本控制](#)之前，使用基于日期的修补程序版本。将此版本视为 EF 4.1 修补程序1(或 EF 4.1.1)。

下面是我们将4.1.1 版本组合在一起的内容列表：

- [EF 4.1.1 Release Post](#)

EF 4。1

EF 4.1.10331 运行时是第一次在 NuGet 上发布，2011年4月。此版本包括简化的 DbContext API 和 Code First 的工作流。

你会注意到奇怪的版本号，4.1.10331，其实际为4.1。此外，还会有一个4.1.10311 版本，该版本应为4.1.0 ("rc" 代表 "候选发布")。我们在决定采用[语义版本控制](#)之前，使用基于日期的修补程序版本。

下面是我们为4.1 版本组合在一起的内容列表。很多情况仍适用于实体框架的更高版本：

- [EF 4.1 发布后](#)
- [Code First 演练](#)
- [Database First 演练 & 模型](#)
- [SQL Azure 联合和实体框架](#)

EF 4。0

此版本包含在2010的4月 .NET Framework 4 和 Visual Studio 2010 中。此版本中的重要新功能包括 POCO 支持、外键映射、延迟加载、可测试性改进、可自定义代码生成和 Model First 工作流。

尽管它是实体框架的第二个版本，但它被命名为 EF 4，以与其随附的 .NET Framework 版本保持一致。在此版本发布后，我们开始在 NuGet 上提供实体框架，并采纳语义版本控制，因为我们不再与 .NET Framework 版本关联。

请注意，某些后续版本的 .NET Framework 附带了包含 EF 位的重大更新。事实上，EF 5.0 的许多新功能已实现为这些位的改进。但是，为了合理化 EF 的版本控制情景，我们将继续引用作为 EF 4.0 运行时的一部分 .NET Framework 的 EF 位，而所有更新版本都包含[EntityFramework NuGet 包](#)。

EF 3。5

实体框架的初始版本包含在3.5 年 8 2008 月发布的 .NET Service Pack 1 和 Visual Studio 2008 SP1 中。此版本提供了使用 Database First 工作流的基本 O/RM 支持。

升级到实体框架6

2020/3/11 ·

在以前版本的 EF 中，代码在作为 NuGet 包附带的 .NET Framework 和带外 (OOB) 库 (主要是 EntityFramework) 的一部分随附的核心库 (主要为 System.web.dll) 中拆分。EF6 采用核心库中的代码，并将其合并到 OOB 库中。此操作是必需的，以便使 EF 成为开源，使其能够以不同于 .NET Framework 的步调发展。这样做的结果是需要针对已移动的类型重新生成应用程序。

对于使用 EF 4.1 和更高版本中随附的 DbContext 的应用程序，这应该非常简单。对于使用 ObjectContext 但仍不难执行的应用程序，还需要执行其他一些工作。

下面是将现有应用程序升级到 EF6 所需执行的操作清单。

1. 安装 EF6 NuGet 包

需要升级到新的实体框架6运行时。

1. 右键单击项目，然后选择“管理 NuGet 包 ...”
2. 在“联机”选项卡中选择 EntityFramework，然后单击“安装”

NOTE

如果安装了 EntityFramework NuGet 包的以前版本，则会将其升级到 EF6。

或者，你可以从包管理器控制台运行以下命令：

```
Install-Package EntityFramework
```

2. 确保删除对 system.string 的程序集引用

安装 EF6 NuGet 包应会自动从你的项目中删除对 System.object 的任何引用。

3. 将任何 EF 设计器(EDMX)模型交换为使用 EF 1.x 代码生成

如果使用 EF 设计器创建了任何模型，则需要更新代码生成模板以生成 EF6 兼容代码。

NOTE

目前只有适用于 Visual Studio 2012 和 2013 的 EF 1.x DbContext 生成器模板。

1. 删除现有代码生成模板。通常，这些文件将命名为 <edmx_file_name>.tt 和 <edmx_file_name>.Context.tt，并将其嵌套在解决方案资源管理器中的 edmx 文件下。可以在解决方案资源管理器中选择模板，然后按 Del 键将它们删除。

NOTE

在网站项目中，模板不会嵌套在 edmx 文件下，但会在解决方案资源管理器中列出。

NOTE

在 VB.NET 项目中，你将需要启用 "显示所有文件" 才能查看嵌套的模板文件。

- 添加适当的 EF 1.x 代码生成模板。在 EF 设计器中打开模型，右键单击设计图面，然后选择 "添加代码生成项 ... "

- 如果使用的是 DbContext API (推荐)，则 "数据" 选项卡下将提供 EF 1.x DbContext 生成器。

NOTE

如果使用的是 Visual Studio 2012，则需要安装 EF 6 工具才能使用此模板。有关详细信息，请参阅[获取实体框架](#)。

- 如果你使用的是 ObjectContext API，则需要选择 "联机" 选项卡，然后搜索 EF EntityObject 生成器。

- 如果对代码生成模板应用了任何自定义，则需要将其重新应用到更新的模板。

4. 更新正在使用的任何 core EF 类型的命名空间

DbContext 和 Code First 类型的命名空间尚未更改。这意味着，对于使用 EF 4.1 或更高版本的许多应用程序，你无需更改任何内容。

之前在 system.exception 中的类型(如 ObjectContext)已移动到新命名空间。这意味着你可能需要更新使用或导入指令，以针对 EF6 进行构建。

命名空间更改的一般规则是将 System.web.* 中的任何类型都移到 "system.string"。换言之，只需插入 Entity。在 System.object 之后。例如：

- EntityException => System.object.Entity.EntityException
- System.web.ObjectContext => System.object.Entity.对象.ObjectContext
- Dataclasses.dll.RelationshipManager => 的数据.Entity.Dataclasses.dll.RelationshipManager

这些类型位于核心命名空间中，因为它们不能直接用于大多数基于 DbContext 的应用程序。作为 DbContext 的一部分的某些类型仍经常用于基于的应用程序，因此尚未移入核心命名空间。这些位置包括：

- EntityState => System.object.实体.EntityState
- Dataclasses.dll.EdmFunctionAttribute => 的数据.DbFunctionAttribute

NOTE

此类已重命名；具有旧名称的类仍然存在并且工作正常，但现在标记为过时。

- EntityFunctions ==> 。DbFunctions

NOTE

此类已重命名；具有旧名称的类仍然存在并且工作正常，但现在标记为已过时。)

- 空间类(例如，DbGeography、DbGeometry)已从 System.web =>。实体。空间

NOTE

System.web 命名空间中的某些类型不是 EF 程序集。这些类型不会移动，因此它们的命名空间保持不变。

Visual Studio 版本

2020/3/11 ·

建议始终使用最新版本的 Visual Studio，因为它包含适用于 .NET、NuGet 和实体框架的最新工具。事实上，实体框架文档中的各种示例和演练都假设你使用的是最新版本的 Visual Studio。

不过，只要考虑一些差异，就有可能使用不同版本的 Visual Studio 的较旧版本实体框架：

Visual Studio 2017 15.7 及更高版本

- 此版本的 Visual Studio 包括实体框架工具和 EF 6.2 运行时的最新版本，不需要执行其他设置步骤。有关这些版本的详细信息，请参阅[新增功能](#)。
- 使用 EF 工具将实体框架添加到新项目将自动添加 EF 6.2 NuGet 包。可以手动安装或升级到联机提供的任何 EF NuGet 包。
- 默认情况下，此版本的 Visual Studio 中可用的 SQL Server 实例为 LocalDB 实例，名为 MSSQLLocalDB。应使用的连接字符串的服务器部分为 "(localdb)\MSSQLLocalDB"。在代码中 C# 指定连接字符串时，请记住使用前缀为 @ 或双反斜杠 "\\\" 的逐字字符串。

Visual Studio 2015 到 Visual Studio 2017 15。6

- 这些版本的 Visual Studio 包括实体框架工具和运行时 ef6.1.3。有关这些版本的详细信息，请参阅[以前的版本](#)。
- 使用 EF 工具向新项目添加实体框架会自动添加 EF ef6.1.3 NuGet 包。可以手动安装或升级到联机提供的任何 EF NuGet 包。
- 默认情况下，此版本的 Visual Studio 中可用的 SQL Server 实例为 LocalDB 实例，名为 MSSQLLocalDB。应使用的连接字符串的服务器部分为 "(localdb)\MSSQLLocalDB"。在代码中 C# 指定连接字符串时，请记住使用前缀为 @ 或双反斜杠 "\\\" 的逐字字符串。

Visual Studio 2013

- 此版本的 Visual Studio 包括实体框架工具和运行时的旧版本。建议使用 Microsoft 下载中心中提供的[安装程序](#) Entity Framework Tools ef6.1.3 升级到。有关这些版本的详细信息，请参阅[以前的版本](#)。
- 使用升级后的 EF 工具向新项目添加实体框架会自动添加 EF ef6.1.3 NuGet 包。可以手动安装或升级到联机提供的任何 EF NuGet 包。
- 默认情况下，此版本的 Visual Studio 中可用的 SQL Server 实例为 LocalDB 实例，名为 MSSQLLocalDB。应使用的连接字符串的服务器部分为 "(localdb)\MSSQLLocalDB"。在代码中 C# 指定连接字符串时，请记住使用前缀为 @ 或双反斜杠 "\\\" 的逐字字符串。

Visual Studio 2012

- 此版本的 Visual Studio 包括实体框架工具和运行时的旧版本。建议使用 Microsoft 下载中心中提供的[安装程序](#) Entity Framework Tools ef6.1.3 升级到。有关这些版本的详细信息，请参阅[以前的版本](#)。
- 使用升级后的 EF 工具向新项目添加实体框架会自动添加 EF ef6.1.3 NuGet 包。可以手动安装或升级到联机提供的任何 EF NuGet 包。
- 默认情况下，此版本的 Visual Studio 中可用的 SQL Server 实例是名为 "v 11.0" 的 LocalDB 实例。应使用的连接字符串的服务器部分为 "(localdb)\v 11.0"。在代码中 C# 指定连接字符串时，请记住使用前缀为 @ 或双反斜杠 "\\\" 的逐字字符串。

Visual Studio 2010

- 此版本的 Visual Studio 中提供的 Entity Framework Tools 版本与实体框架6运行时不兼容，因此无法升级。
- 默认情况下，实体框架工具会将实体框架4.0 添加到你的项目。若要使用任何更高版本的 EF 创建应用程序，首先需要安装[NuGet 包管理器扩展](#)。
- 默认情况下，EF 工具版本中的所有代码生成都基于 EntityObject 和实体框架4。建议通过安装[C#或Visual Basic](#)的 DbContext 代码生成模板，将代码生成切换为基于 DbContext 和实体框架5。
- 安装 NuGet 包管理器扩展后，可以手动安装或升级到任何联机可用的 EF NuGet 包，并将 EF6 与 Code First 一起使用，这不需要设计器。
- 默认情况下，此版本的 Visual Studio 中提供的 SQL Server 实例 SQL Server Express 称为 SQLEXPRESS。应使用的连接字符串的服务器部分是 ""。\\SQLEXPRESS "。在代码中C#指定连接字符串时，请记住使用前缀为 @ 或双反斜杠 "\\\" 的逐字字符串。

开始使用 Entity Framework 6

2020/4/8 • [Edit Online](#)

本指南包含指向精选文章、演练和视频的链接集合，这些内容可帮助你快速入门。

基础知识

- [获取 Entity Framework](#)

此处可学习如何将 Entity Framework 添加到应用程序。如果要使用 EF 设计器，请确保在 Visual Studio 中安装它。

- [创建模型:Code First、EF 设计器和 EF 工作流](#)

是否希望指定 EF 模型编写代码或绘制方框和线条？是否要使用 EF 来将对象映射到现有数据库，或希望 EF 创建为对象量身打造的数据库？本文介绍了使用 EF6 的两种不同方法：EF 设计器和 Code First。请确保关注讨论内容并查看有关不同之处的视频。

- [使用 DbContext](#)

DbContext 是需要学习其使用方法的第一个也是最重要的一个 EF 类型。它可用作数据库查询的启动板，并可跟踪对对象作出的更改，以便持续存回数据库。

- [提出问题](#)

了解如何获取专家的帮助，并向社区贡献自己的答案。

- [参与](#)

Entity Framework 6 采用开放式开发模型。访问我们的 GitHub 存储库，了解如何帮助改进 EF。

Code First 资源

- [对现有数据库工作流采用 Code First](#)
- [对新的数据库工作流采用 Code First](#)
- [使用 Code First 映射枚举](#)
- [使用 Code First 映射空间类型](#)
- [编写自定义 Code First 约定](#)
- [将 Code First Fluent 配置与 Visual Basic 配合使用](#)
- [Code First 迁移](#)
- [团队环境中的 Code First 迁移](#)
- [自动 Code First 迁移\(不再推荐\)](#)

EF 设计器资源

- [Database First 工作流](#)
- [Model First 工作流](#)
- [映射枚举](#)
- [映射空间类型](#)
- [每个层次结构一张表继承映射](#)
- [每个类型一张表继承映射](#)

- [用于更新的存储过程映射](#)
- [用于查询的存储过程映射](#)
- [实体拆分](#)
- [表拆分](#)
- [定义查询\(高级\)](#)
- [表值函数\(高级\)](#)

其他资源

- [异步查询和保存](#)
- [使用 WinForms 进行数据绑定](#)
- [使用 WPF 进行数据绑定](#)
- [使用自跟踪实体的断开连接的方案\(不再推荐\)](#)

获取实体框架

2020/3/11 •

实体框架由适用于 Visual Studio 的 EF 工具和 EF 运行时组成。

适用于 Visual Studio 的 EF 工具

Visual Studio 的 Entity Framework Tools 包括 EF 设计器和 EF 模型向导，并是数据库优先和模型优先工作流所必需的。所有最新版本的 Visual Studio 中都包含 EF 工具。如果执行 Visual Studio 的自定义安装，则需要确保选择项“实体框架6工具”，方法是选择包含它的工作负载，或将其选择为单个组件。

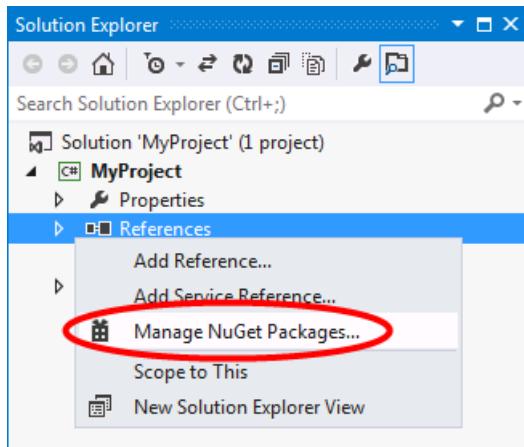
对于某些早期版本的 Visual Studio，更新的 EF 工具可作为下载。请参阅[Visual Studio 版本](#)，了解有关如何获取适用于你的 Visual Studio 版本的最新版本 EF 工具的指南。

EF 运行时

实体框架的最新版本可作为[EntityFramework NuGet 包](#)提供。如果你不熟悉 NuGet 包管理器，我们建议你阅读[NuGet 概述](#)。

安装 EF NuGet 包

可以通过右键单击项目的“引用”文件夹并选择“管理 NuGet 包...”来安装 EntityFramework 包。



从包管理器控制台安装

或者，你可以通过在[包管理器控制台](#)中运行以下命令来安装 EntityFramework。

```
Install-Package EntityFramework
```

安装特定版本的 EF

从 EF 4.1 开始，已发布了 EF 运行时的新版本作为[EntityFramework NuGet 包](#)。可以通过在 Visual Studio 的[包管理器控制台](#)中运行以下命令，将这些版本中的任何一种添加到基于 .NET Framework 的项目中：

```
Install-Package EntityFramework -Version <number>
```

请注意，`<number>` 表示要安装的 EF 的特定版本。例如，6.2.0 是 EF 6.2 的编号版本。

4.1 之前的 EF 运行时是 .NET Framework 的一部分，不能单独安装。

安装最新预览版

上述方法将为你介绍实体框架的最新完全受支持的版本。实体框架提供了预发布版本的预发布版本，我们希望你尝试并向我们提供反馈。

若要安装最新的 EntityFramework 预览版，可以在 "管理 NuGet 包" 窗口中选择 "包括预发行版"。如果没有可用的预发布版本，则会自动获取实体框架的最新完全受支持版本。



或者，你可以在[包管理器控制台](#)中运行以下命令。

```
Install-Package EntityFramework -Pre
```

使用 DbContext

2020/3/11 ·

若要使用实体框架来使用 .NET 对象查询、插入、更新和删除数据，您首先需要创建一个模型，该模型将模型中定义的实体和关系映射到数据库中的表。

有了模型后，应用程序与之交互的主类就 `System.Data.Entity.DbContext`（通常称为上下文类）。您可以使用与模型关联的 `DbContext` 来执行以下操作：

- 编写和执行查询
- 将查询结果具体化为实体对象
- 跟踪对这些对象进行的更改
- 将对象更改保存回数据库
- 将内存中的对象绑定到 UI 控件

本页提供有关如何管理上下文类的一些指导。

定义 DbContext 派生类

使用上下文的建议方法是定义从 `DbContext` 派生的类，并公开 `DbSet` 属性，这些属性表示上下文中指定实体的集合。如果使用的是 EF 设计器，则会为您生成上下文。如果使用 Code First，则通常会自行编写上下文。

```
public class ProductContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
```

拥有上下文后，可以通过这些属性查询、添加（使用 `Add` 或 `Attach` 方法）或在上下文中删除（使用 `Remove`）实体。访问上下文对象上的 `DbSet` 属性表示一个启动查询，该查询返回指定类型的所有实体。请注意，仅访问属性不会执行查询。执行以下操作时执行查询：

- `foreach` (C#) 或 `For Each` (Visual Basic) 语句枚举对象。
- 它通过集合操作（如 `ToArray`、`ToDictionary` 或 `ToList`）进行枚举。
- 在查询的最外面部分指定 LINQ 运算符，例如 `First` 或 `Any`。
- 如果在上下文中未找到具有指定键的实体，则调用以下方法之一：`Load` 扩展方法 `DbEntityEntry.Reload`、`Database.ExecuteSqlCommand` 和 `DbSet<T>.Find`。

生存期

上下文的生存期在实例创建时开始，并在实例被释放或被回收时结束。如果希望在块的末尾释放上下文控制的所有资源，请使用。使用时，编译器会自动创建 `try/finally` 块并在 `finally` 块中调用 `dispose`。

```
public void UseProducts()
{
    using (var context = new ProductContext())
    {
        // Perform data access using the context
    }
}
```

下面是决定上下文生存期时的一些一般指导原则：

- 使用 Web 应用程序时，请为每个请求使用上下文实例。
- 使用 Windows Presentation Foundation (WPF) 或 Windows 窗体时，请使用每个窗体的上下文实例。这使你可以使用上下文提供的更改跟踪功能。
- 如果上下文实例是由依赖关系注入容器创建的，则该容器通常负责释放上下文。
- 如果上下文是在应用程序代码中创建的，请记得在不再需要时释放上下文。
- 使用长时间运行的上下文时，请考虑以下事项：
 - 在将更多对象及其引用加载到内存中时，上下文的内存消耗可能会迅速增加。这可能会导致性能问题。
 - 上下文不是线程安全的，因此不应在多个线程上同时对其执行工作。
 - 如果异常导致上下文处于不可恢复的状态，则整个应用程序可能会终止。
 - 随着查询数据的时间和更新数据的时间的差距增大，出现与并发性相关的问题的可能性将会增加。

连接

默认情况下，上下文管理与数据库的连接。上下文会根据需要打开和关闭连接。例如，上下文打开一个连接来执行查询，然后在处理完所有结果集后关闭连接。

在某些情况下，您需要加强控制应在哪些情况下打开和关闭连接。例如，使用 SQL Server Compact 时，通常建议在应用程序的生存期内维护单独的数据库打开连接，以提高性能。您可以使用 `Connection` 属性手动管理此过程。

关系、导航属性和外键

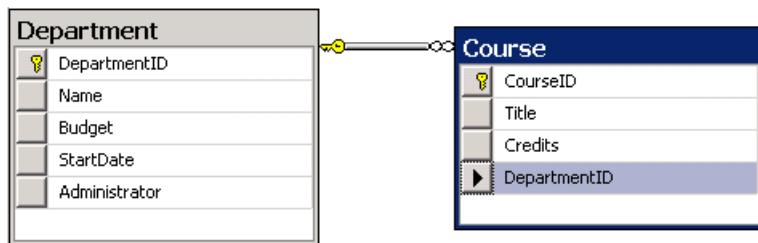
2020/3/11 •

本文概述了实体框架如何管理实体之间的关系。它还提供了一些有关如何映射和操作关系的指导。

EF 中的关系

在关系数据库中，表之间的关系（也称为关联）通过外键定义。外键（FK）是用于建立和加强两个表数据之间的链接的一列或多列。通常有三种类型的关系：一对一、一对多和多对多的关系。在一对多关系中，外键在表示关系的多个端的表上定义。多对多关系涉及定义第三个表（称为“联接表”或“联接表”），其主键由这两个相关表中的外键构成。在一一对多关系中，主键除了作为外键以外，对于任何一个表都没有单独的外键列。

下图显示了参与一对多关系的两个表。课程表是依赖表，因为它包含将其链接到部门表的DepartmentID列。



在实体框架中，可以通过关联或关系将实体与其他实体相关。每个关系都包含两个端，用于描述该关系中的两个实体的实体类型和类型（一、零或多个）的重数。关系可能由引用约束控制，这描述了关系中的哪一端是主体角色，后者是依赖角色。

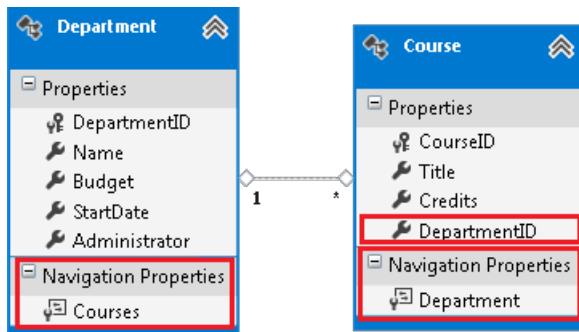
导航属性提供了一种在两个实体类型之间导航关联的方法。针对对象参与到其中的每个关系，各对象均可以具有导航属性。使用导航属性，您可以在两个方向上导航和管理关系，返回引用对象（如果重数为1或零）或集合（如果重数为多个）。您还可以选择使用单向导航，在这种情况下，只需在参与关系的一种类型上定义导航属性，而不是在两者上定义。

建议在模型中包含映射到数据库中的外键的属性。加入了外键属性，您就可以通过修改依赖对象的外键值来创建或更改关系。此类关联称为外键关联。在处理断开连接的实体时，使用外键更为重要。请注意，当使用1对1或1到0时，请注意。1关系，没有单独的外键列，主键属性充当外键，并且始终包含在模型中。

如果模型中不包含外键列，则会将关联信息作为独立对象进行管理。关系通过对象引用而不是外键属性进行跟踪。这种类型的关联称为独立关联。修改独立关联的最常见方法是修改为参与关联的每个实体生成的导航属性。

可以在您的模型中选择使用一种或两种类型的关联。但是，如果您具有通过仅包含外键的联接表连接的一种纯多对多关系，则 EF 将使用独立的关联来管理这类多对多关系。

下图显示了使用 Entity Framework Designer 创建的概念模型。该模型包含两个参与一对多关系的实体。这两个实体都具有导航属性。当然是依赖实体，并且定义了DepartmentID外键属性。



下面的代码片段显示了用 Code First 创建的同一模型。

```
public class Course
{
    public int CourseID { get; set; }
    public string Title { get; set; }
    public int Credits { get; set; }
    public int DepartmentID { get; set; }
    public virtual Department Department { get; set; }
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public DateTime StartDate { get; set; }
    public int? Administrator { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}
```

配置或映射关系

本页的其余部分介绍如何使用关系访问和操作数据。有关设置模型中的关系的信息，请参阅以下页面。

- 若要在 Code First 中配置关系，请参阅[数据批注和熟知的 API-关系](#)。
- 若要使用 Entity Framework Designer 配置关系，请参阅[与 EF 设计器之间的关系](#)。

创建和修改关系

在外键关联中，当你更改关系时，具有 `EntityState.Unchanged` 状态的依赖对象的状态将更改为 " `EntityState.Modified`"。在独立关系中，更改关系不会更新依赖对象的状态。

下面的示例演示如何使用外键属性和导航属性将相关对象关联起来。使用外键关联，可以使用这两种方法更改、创建或修改关系。使用独立关联，则不能使用外键属性。

- 将新值分配给外键属性，如下面的示例中所示。

```
course.DepartmentID = newCourse.DepartmentID;
```

- 下面的代码通过将外键设置为`null`来移除关系。请注意，外键属性必须可以为 `null`。

```
course.DepartmentID = null;
```

NOTE

如果引用处于添加状态(在本示例中为课程对象)，则在调用 `SaveChanges` 之前，引用导航属性将不会与新对象的键值同步。由于对象上下文在键值保存前不包含已添加对象的永久键，因此不发生同步。如果在设置关系后，必须完全同步新对象，请使用以下方法之一。

- 通过将一个新对象分配给导航属性。下面的代码创建课程与 `department` 之间的关系。如果将对象附加到上下文，则还会将 `course` 添加到 `department.Courses` 集合，`course` 对象上的相应外键属性将设置为该部门

的键属性值。

```
course.Department = department;
```

- 若要删除关系, 请将导航属性设置为 `null`。如果使用的是基于 .NET 4.0 的实体框架, 则需要加载相关端, 然后将其设置为 `null`。例如:

```
context.Entry(course).Reference(c => c.Department).Load();
course.Department = null;
```

从基于 .NET 4.5 实体框架 5.0 开始, 可以在不加载相关端的情况下将关系设置为 `null`。还可以使用以下方法将当前值设置为 `null`。

```
context.Entry(course).Reference(c => c.Department).CurrentValue = null;
```

- 通过在实体集合中删除或添加对象。例如, 可以将类型 `Course` 的对象添加到 `department.Courses` 集合。此操作创建特定课程和特定 `department` 之间的关系。如果将对象附加到上下文, 则会将课程对象上的部门引用和外键属性设置为相应的 `department`。

```
department.Courses.Add(newCourse);
```

- 通过使用 `ChangeRelationshipState` 方法来更改两个实体对象之间的指定关系的状态。此方法最常用于使用 N 层应用程序和独立关联(不能与外键关联一起使用)。此外, 若要使用此方法, 你必须下拉 `ObjectContext`, 如以下示例中所示。

在下面的示例中, 讲师和课程之间存在多对多的关系。调用 `ChangeRelationshipState` 方法并传递 `EntityState.Added` 参数, 可让 `SchoolContext` 了解已在两个对象之间添加了关系:

```
((IObjectContextAdapter)context).ObjectContext.
    ObjectStateManager.
    ChangeRelationshipState(course, instructor, c => c.Instructor, EntityState.Added);
```

请注意, 如果您要更新关系, 则必须在添加新关系后删除旧关系:

```
((IObjectContextAdapter)context).ObjectContext.
    ObjectStateManager.
    ChangeRelationshipState(course, oldInstructor, c => c.Instructor, EntityState.Deleted);
```

同步外键和导航属性之间的更改

使用上述方法之一更改附加到上下文的对象的关系时, 实体框架需要使外键、引用和集合保持同步。实体框架会自动为具有代理的 POCO 实体管理此同步(也称为关系修补)。有关详细信息, 请参阅[使用代理](#)。

如果使用不带代理的 POCO 实体, 则必须确保调用 `DetectChanges` 方法来同步上下文中的相关对象。请注意, 以下 API 会自动触发 `DetectChanges` 调用。

- `DbSet.Add`
- `DbSet.AddRange`
- `DbSet.Remove`
- `DbSet.RemoveRange`
- `DbSet.Find`

- `DbSet.Local`
- `DbContext.SaveChanges`
- `DbSet.Attach`
- `DbContext.GetValidationErrors`
- `DbContext.Entry`
- `DbChangeTracker.Entries`
- 对 `DbSet` 执行 LINQ 查询

正在加载相关对象

在实体框架通常使用导航属性来加载通过定义的关联与返回的实体相关的实体。有关详细信息，请参阅[加载相关对象](#)。

NOTE

在外键关联中，加载依赖对象的相关端时，将会基于当前位于内存中的依赖对象的外键值加载相关对象：

```
// Get the course where currently DepartmentID = 2.  
Course course2 = context.Courses.First(c=>c.DepartmentID == 2);  
  
// Use DepartmentID foreign key property  
// to change the association.  
course2.DepartmentID = 3;  
  
// Load the related Department where DepartmentID = 3  
context.Entry(course).Reference(c => c.Department).Load();
```

在独立关联中，基于当前数据库中的外键值查询依赖对象的相关端。但是，如果关系已修改，且依赖对象的引用属性指向对象上下文中加载的其他主体对象，实体框架将尝试创建关系，因为它是在客户端上定义的。

管理并发

在外键和独立关联中，**并发**检查基于在模型中定义的实体键和其他实体属性。使用 EF 设计器创建模型时，请将 `ConcurrencyMode` 特性设置为 `fixed`，以指定应检查属性的并发性。使用 Code First 定义模型时，请对要检查其并发性的属性使用 `ConcurrencyCheck` 批注。使用 Code First 时，还可以使用 `TimeStamp` 批注来指定应检查属性的并发性。给定类中只能有一个时间戳属性。Code First 将此属性映射到数据库中不可以为 null 的字段。

建议你始终在使用参与并发检查和解析的实体时使用外键关联。

有关详细信息，请参阅[处理并发冲突](#)。

使用重叠键

重叠键是指键中某些属性亦是实体中其他键的一部分的那些组合键。在独立关联中不能包含重叠键。若要更改包含重叠键的外键关联，我们建议您修改外键值而不要使用对象引用。

异步查询并保存

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

EF6 引入了对异步查询的支持，并使用 .NET 4.5 中引入的 `async` 和 `await` 关键字进行保存。虽然并非所有应用程序都可以受益于异步，但它可用于在处理长时间运行的、网络或 i/o 限制的任务时提高客户端的响应能力和服务器的可伸缩性。

何时真正使用 `async`

本演练的目的是以一种可以轻松地观察异步和同步程序执行之间的差异的方式引入异步概念。本演练并不旨在说明异步编程提供了好处的关键方案。

异步编程主要侧重于释放当前托管线程（运行 .NET 代码的线程）来执行其他工作，同时等待不需要托管线程的任何计算时间的操作。例如，当数据库引擎正在处理查询时，.NET 代码不会执行任何操作。

在客户端应用程序（WinForms、WPF 等）中，当前线程可用于在执行异步操作时保持 UI 的响应能力。在服务器应用程序（ASP.NET 等）中，线程可用于处理其他传入请求-这可以减少内存使用量和/或提高服务器的吞吐量。

在大多数使用 `async` 的应用程序中，没有明显的好处，甚至可能会造成不利影响。在提交到特定方案之前，请使用测试、分析和常见意义来度量异步的影响。

下面是一些用于了解 `async` 的更多资源：

- [.NET 4.5 中的 Brandon Bray 概述](#)
- MSDN Library 中的[异步编程](#)页
- [如何使用 Async 构建 ASP.NET Web 应用程序](#)（包括提高服务器吞吐量的演示）

创建模型

我们将使用[Code First 工作流](#)来创建模型并生成数据库，但异步功能适用于所有 ef 模型，包括使用 EF 设计器创建的那些模型。

- 创建控制台应用程序并将其调用 `AsyncDemo`
- 添加 EntityFramework NuGet 包
 - 在解决方案资源管理器中，右键单击 `AsyncDemo` 项目
 - 选择 “[管理 NuGet 包 ...](#)”
 - 在 “[管理 NuGet 包](#)” 对话框中，选择 “[联机](#)” 选项卡，然后选择 “[EntityFramework](#)” 包
 - 单击 “[安装](#)”
- 添加具有以下实现的 `Model.cs` 类

```
using System.Collections.Generic;
using System.Data.Entity;

namespace AsyncDemo
{
    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}
```

创建同步程序

现在我们有了 EF 模型，接下来我们编写一些代码，用它来执行某些数据访问。

- 将Program.cs的内容替换为以下代码

```

using System;
using System.Linq;

namespace AsyncDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            PerformDatabaseOperations();

            Console.WriteLine("Quote of the day");
            Console.WriteLine(" Don't worry about the world coming to an end today... ");
            Console.WriteLine(" It's already tomorrow in Australia.");

            Console.WriteLine();
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        public static void PerformDatabaseOperations()
        {
            using (var db = new BloggingContext())
            {
                // Create a new blog and save it
                db.Blogs.Add(new Blog
                {
                    Name = "Test Blog #" + (db.Blogs.Count() + 1)
                });
                Console.WriteLine("Calling SaveChanges.");
                db.SaveChanges();
                Console.WriteLine("SaveChanges completed.");

                // Query for all blogs ordered by name
                Console.WriteLine("Executing query.");
                var blogs = (from b in db.Blogs
                            orderby b.Name
                            select b).ToList();

                // Write all blogs out to Console
                Console.WriteLine("Query completed with following results:");
                foreach (var blog in blogs)
                {
                    Console.WriteLine(" " + blog.Name);
                }
            }
        }
    }
}

```

此代码调用**PerformDatabaseOperations**方法，该方法将新的**博客**保存到数据库，然后从数据库中检索所有**博客**并将其打印到控制台。完成此操作后，程序将一天的报价写入控制台。

由于代码是同步的，因此，当我们运行程序时，可以观察到以下执行流：

1. **SaveChanges**开始将新**博客**推送到数据库
2. **SaveChanges**完成
3. 所有**博客**的查询都发送到数据库
4. 查询返回并将结果写入控制台
5. 将日报价写入控制台

```
Calling SaveChanges.  
SaveChanges completed.  
Executing query.  
Query completed with following results:  
- Test Blog #1  
Quote of the day  
Don't worry about the world coming to an end today...  
It's already tomorrow in Australia.  
Press any key to exit...
```

使其异步

现在，我们已启动并运行程序，接下来可以开始使用新的 `async` 和 `await` 关键字。我们已对 `Program.cs` 进行了以下更改

1. 第2行: `system.web`命名空间的 `using` 语句使我们能够访问 EF `async` extension 方法。
2. 第4行: 用于`system.web`命名空间的 `using` 语句允许使用任务类型。
3. 第12行 & 18: 我们正在捕获作为任务，用于监视`PerformSomeDatabaseOperations` (第12行) 的进度，然后在程序的所有工作完成后阻止程序执行完成此任务完成(第18行)。
4. 第25行: 我们更新了`PerformSomeDatabaseOperations`，将其标记为`async`并返回一个任务。
5. 第35行: 现在，我们正在调用 `SaveChanges` 的异步版本，并等待其完成。
6. 第42行: 我们正在调用 `System.Linq.Enumerable.ToList` 的异步版本，并等待结果。

有关 `System.web` 命名空间中可用扩展方法的完整列表，请参阅 `QueryableExtensions` 类。还需要向 `using` 语句添加 "使用 `System.web`"。

```

using System;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace AsyncDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            var task = PerformDatabaseOperations();

            Console.WriteLine("Quote of the day");
            Console.WriteLine(" Don't worry about the world coming to an end today... ");
            Console.WriteLine(" It's already tomorrow in Australia.");

            task.Wait();

            Console.WriteLine();
            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        public static async Task PerformDatabaseOperations()
        {
            using (var db = new BloggingContext())
            {
                // Create a new blog and save it
                db.Blogs.Add(new Blog
                {
                    Name = "Test Blog #" + (db.Blogs.Count() + 1)
                });
                Console.WriteLine("Calling SaveChanges.");
                await db.SaveChangesAsync();
                Console.WriteLine("SaveChanges completed.");

                // Query for all blogs ordered by name
                Console.WriteLine("Executing query.");
                var blogs = await (from b in db.Blogs
                                   orderby b.Name
                                   select b).ToListAsync();

                // Write all blogs out to Console
                Console.WriteLine("Query completed with following results:");
                foreach (var blog in blogs)
                {
                    Console.WriteLine(" - " + blog.Name);
                }
            }
        }
    }
}

```

由于代码是异步的，因此，我们可以在运行程序时观察到不同的执行流程：

- 1. SaveChanges开始将新博客推送到数据库**

将命令发送到数据库后，当前托管线程不需要更多计算时间。*PerformDatabaseOperations*方法返回（即使尚未执行完毕）和*Main*方法中的程序流将继续。

- 2. 将日报价写入控制台**

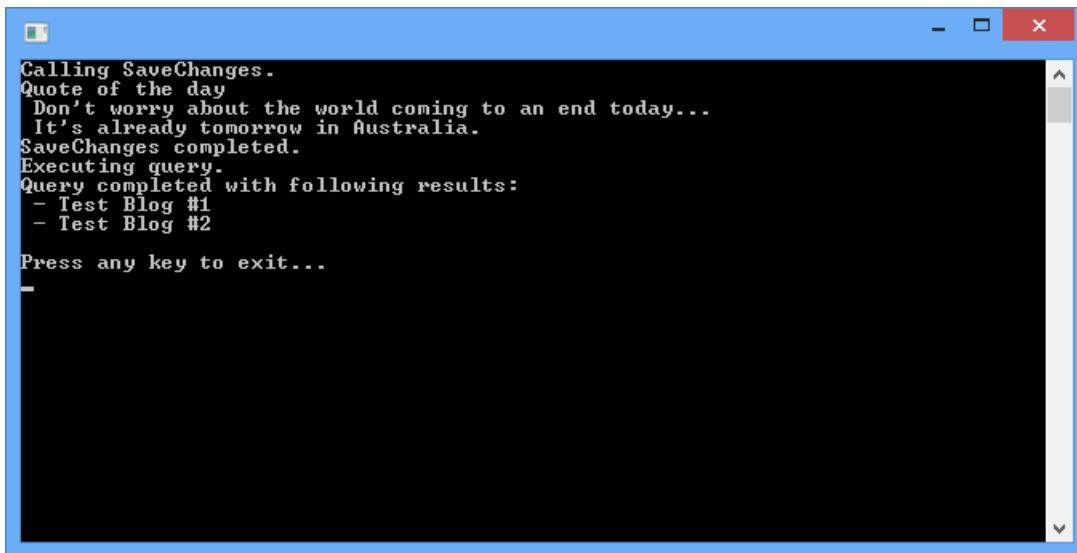
由于在*Main*方法中没有更多的工作要做，因此，在数据库操作完成前，会阻止在等待调用上托管线程。完成后，我们将执行*PerformDatabaseOperations*的剩余部分。

- 3. SaveChanges完成**

- 4. 所有博客的查询都发送到数据库**

同样，在数据库中处理查询时，托管线程可以自由地执行其他工作。由于所有其他执行都已完成，线程只会在等待调用时停止。

5. 查询返回并将结果写入控制台



```
Calling SaveChanges.
Quote of the day
Don't worry about the world coming to an end today...
It's already tomorrow in Australia.
SaveChanges completed.
Executing query.
Query completed with following results:
- Test Blog #1
- Test Blog #2

Press any key to exit...
```

要点在于

现在，我们看到了使用 EF 的异步方法是多么简单。尽管 `async` 的优点在简单的控制台应用程序中可能并不是很明显，但在运行长时间运行或网络绑定的活动可能会阻止应用程序或导致大量线程增加内存占用量。

基于代码的配置

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

可以在配置文件 (app.config/web.config) 或代码中指定实体框架应用程序的配置。后者称为基于代码的配置。

在[单独的文章](#)中介绍了配置文件中的配置。配置文件优先于基于代码的配置。换句话说，如果在代码和配置文件中都设置了配置选项，则使用配置文件中的设置。

使用 DbConfiguration

EF6 和更高版本中基于代码的配置是通过创建 `DbConfiguration` 的子类来实现的。在对 `DbConfiguration` 进行子类化时，应遵循以下准则：

- 只为应用程序创建一个 `DbConfiguration` 类。此类指定应用域范围内的设置。
- 将 `DbConfiguration` 类放在与 `DbContext` 类相同的程序集中。（若要更改此内容，请参阅[移动 `DbConfiguration` 部分](#)。）
- 为 `DbConfiguration` 类指定一个公共的无参数构造函数。
- 通过在此构造函数中调用受保护的 `DbConfiguration` 方法来设置配置选项。

遵循这些指导原则后，EF 可以通过需要访问模型和应用程序运行时的两个工具自动发现和使用配置。

示例

派生自 `DbConfiguration` 的类可能如下所示：

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;

namespace MyNamespace
{
    public class MyConfiguration : DbConfiguration
    {
        public MyConfiguration()
        {
            SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
            SetDefaultConnectionFactory(new LocalDbConnectionFactory("mssqllocaldb"));
        }
    }
}
```

此类设置 EF 以使用 SQL Azure 执行策略-自动重试失败的数据库操作，并使用由 Code First 中的约定创建的数据库的本地数据库。

移动 `DbConfiguration`

有些情况下，不能将 `DbConfiguration` 类放在与 `DbContext` 类相同的程序集中。例如，你可能在不同的程序集中有两个 `DbContext` 类。有两个选项可用于处理此操作。

第一种方法是使用配置文件指定要使用的 DbConfiguration 实例。为此, 请设置 entityFramework 部分的 codeConfigurationType 属性。例如:

```
<entityFramework codeConfigurationType="MyNamespace.MyDbConfiguration, MyAssembly">
    ...
</entityFramework>
```

CodeConfigurationType 的值必须是 DbConfiguration 类的程序集和命名空间限定名称。

第二种方法是在上下文类上放置 DbConfigurationTypeAttribute。例如:

```
[DbConfigurationType(typeof(MyDbConfiguration))]
public class MyContextContext : DbContext
{
}
```

传递给特性的值可以是 DbConfiguration 类型(如上所示), 也可以是程序集和命名空间限定的类型名称字符串。例如:

```
[DbConfigurationType("MyNamespace.MyDbConfiguration, MyAssembly")]
public class MyContextContext : DbContext
{
}
```

显式设置 DbConfiguration

在某些情况下, 在使用任何 DbContext 类型之前可能需要进行配置。这种情况的示例包括:

- 在不使用上下文的情况下使用 DbModelBuilder 生成模型
- 在使用应用程序上下文之前, 使用一些其他使用 DbContext 的框架/实用工具代码, 其中使用了该上下文

在这种情况下, EF 无法自动发现配置, 而必须执行以下操作之一:

- 设置配置文件中的 DbConfiguration 类型, 如上面的 [移动 DbConfiguration](#) 部分中所述
- 在应用程序启动过程中调用静态 DbConfiguration SetConfiguration 方法

重写 DbConfiguration

在某些情况下, 需要重写 DbConfiguration 中的配置集。这通常不是由应用程序开发人员完成, 而是由不能使用派生的 DbConfiguration 类的第三方提供程序和插件来完成。

为此, EntityFramework 允许注册事件处理程序, 该事件处理程序可以在锁定之前修改现有配置。它还提供专门用于替换 EF 服务定位器返回的任何服务的一个糖方法。这就是如何使用它:

- 在应用启动时(使用 EF 之前)插件或提供程序应为此事件注册事件处理程序方法。(请注意, 此操作必须在应用程序使用 EF 之前发生。)
- 对于需要替换的每个服务, 事件处理程序都会调用 ReplaceService。

例如, 若要替换 IDbConnectionFactory 和 DbProviderService, 需注册一个类似于以下内容的处理程序:

```
DbConfiguration.Loaded += (_, a) =>
{
    a.ReplaceService<DbProviderServices>((s, k) => new MyProviderServices(s));
    a.ReplaceService<IDbConnectionFactory>((s, k) => new MyConnectionFactory(s));
};
```

在上面的代码中，`MyProviderServices` 和 `MyConnectionFactory` 表示服务的实现。

你还可以添加其他依赖关系处理程序以获得相同的效果。

请注意，你也可以通过这种方式来包装 `DbProviderFactory`，但这样做只会影响 EF，而不会在 EF 之外使用 `DbProviderFactory`。出于此原因，你可能需要继续像以前一样来包装 `DbProviderFactory`。

你还应记住你在应用程序外部运行的服务（例如，从包管理器控制台运行迁移时）。当你从控制台运行迁移时，它将尝试查找你的 `DbConfiguration`。但是，无论是否获取包装服务，都取决于它所注册的事件处理程序的位置。如果已将其注册为 `DbConfiguration` 构造的一部分，则应执行代码，并将服务打包。通常不会出现这种情况，这意味着工具不会获得包装服务。

配置文件设置

2020/3/11 •

实体框架允许在配置文件中指定多个设置。在常规 EF 中，遵循 "约定 over 配置" 原则：在此文章中讨论的所有设置都有默认行为，你只需考虑在默认不再满足你的要求时更改设置。

基于代码的替代项

还可以使用代码应用所有这些设置。从 EF6 开始，我们引入了[基于代码的配置](#)，它提供了一种从代码应用配置的集中方式。在 EF6 之前，仍可以从代码应用配置，但需要使用各种 API 来配置不同的区域。配置文件选项允许在部署期间轻松地更改这些设置，而无需更新代码。

"实体框架配置" 部分

从 EF 4.1 开始，你可以使用配置文件的 **appSettings** 节为上下文设置数据库初始值设定项。在 EF 4.3 中，我们引入了自定义 **entityFramework** 部分来处理新设置。实体框架仍将识别使用旧格式设置的数据库初始值设定项，但建议在可能的情况下移动到新的格式。

安装 EntityFramework NuGet 包时，**entityFramework** 节会自动添加到项目的配置文件中。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.3.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </configSections>
</configuration>
```

连接字符串

[本页提供了](#)有关实体框架如何确定要使用的数据库的更多详细信息，包括配置文件中的连接字符串。

连接字符串位于标准 **connectionStrings** 元素中，不需要 **entityFramework** 部分。

基于 Code First 的模型使用常规 ADO.NET 连接字符串。例如：

```
<connectionStrings>
  <add name="BlogContext"
    providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Blogging;Integrated Security=True;" />
</connectionStrings>
```

基于 EF 设计器的模型使用特殊 EF 连接字符串。例如：

```
<connectionStrings>
  <add name="BlogContext"
    connectionString=
      "metadata=
        res://*/BloggingModel.csdl|
        res://*/BloggingModel.ssdl|
        res://*/BloggingModel.msl;
      provider=System.Data.SqlClient;
      provider connection string=
        "data source=(localdb)\mssqllocaldb;
        initial catalog=Blogging;
        integrated security=True;
        multipleactiveresultsets=True;"
      providerName="System.Data.EntityClient" />
</connectionStrings>
```

基于代码的配置类型 (EF6)

从 EF6 开始，可以指定 EF 的 DbConfiguration，以便在应用程序中使用[基于代码的配置](#)。在大多数情况下，无需指定此设置，因为 EF 会自动发现你的 DbConfiguration。有关可能需要在配置文件中指定 DbConfiguration 的详细信息，请参阅[基于代码的配置的移动 DbConfiguration](#)部分。

若要设置 DbConfiguration 类型，请在codeConfigurationType元素中指定程序集限定的类型名称。

NOTE

程序集限定名称是命名空间限定名称，后跟一个逗号，然后是该类型所在的程序集。还可以选择指定程序集版本、区域性和公钥标记。

```
<entityFramework codeConfigurationType="MyNamespace.MyConfiguration, MyAssembly">
</entityFramework>
```

EF 数据库提供程序 (EF6)

在 EF6 之前，数据库提供程序实体框架特定部分必须作为核心 ADO.NET 提供程序的一部分包括在内。从 EF6 开始，EF 特定部分现在被管理和注册。

通常无需自行注册提供程序。此操作通常会在安装时由提供程序完成。

提供程序是通过在entityFramework 节的 provider 子节下包含提供程序元素来注册的。提供程序条目有两个必需的属性：

- invariantName标识此 EF 提供程序针对的核心 ADO.NET 提供程序
- 类型是 EF 提供程序实现的程序集限定类型名称

NOTE

程序集限定名称是命名空间限定名称，后跟一个逗号，然后是该类型所在的程序集。还可以选择指定程序集版本、区域性和公钥标记。

例如，下面是在安装实体框架时创建的用于注册默认 SQL Server 提供程序的条目。

```
<providers>
  <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices,
EntityFramework.SqlServer" />
</providers>
```

拦截(EF 6.1 以上版本)

从 EF 6.1 开始，可以在配置文件中注册侦听器。当 EF 执行某些操作(例如执行数据库查询、打开连接等)时，拦截允许您运行其他逻辑。

通过在entityFramework部分的拦截程序子部分下包含一个侦听器元素来注册拦截程序。例如，下面的配置将注册将所有数据库操作记录到控制台的内置DatabaseLogger侦听器。

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework"/>
</interceptors>
```

将数据库操作记录到文件中(EF 6.1 以上版本)

如果要将日志记录添加到现有的应用程序以帮助调试问题，则通过配置文件注册侦听器特别有用。

DatabaseLogger支持通过将文件名作为构造函数参数提供来记录文件。

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Temp\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

默认情况下，每次启动应用程序时，将会使用新文件覆盖日志文件。如果日志文件已存在，则改为将其追加到日志文件中，如下所示：

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Temp\LogOutput.txt"/>
      <parameter value="true" type="System.Boolean"/>
    </parameters>
  </interceptor>
</interceptors>
```

有关DatabaseLogger和注册侦听器的其他信息，请参阅博客文章[EF 6.1：在不重新编译的情况下启用日志记录](#)。

Code First 默认连接工厂

"配置" 部分允许你指定 Code First 应用于查找要用于上下文的数据库的默认连接工厂。仅当未将连接字符串添加到上下文的配置文件时，才使用默认连接工厂。

当你安装 EF NuGet 包时，将根据你安装的是 SQL Express 或 LocalDB 来注册一个默认连接工厂。

若要设置连接工厂，请在defaultConnectionFactory元素中指定程序集限定的类型名称。

NOTE

程序集限定名称是命名空间限定名称，后跟一个逗号，然后是该类型所在的程序集。还可以选择指定程序集版本、区域性和公钥标记。

下面是设置自己的默认连接工厂的示例：

```
<entityFramework>
  <defaultConnectionFactory type="MyNamespace.MyCustomFactory, MyAssembly"/>
</entityFramework>
```

上面的示例要求自定义工厂具有无参数的构造函数。如果需要，可以使用`parameters`元素指定构造函数参数。

例如，`SqlCeConnectionFactory` 包含在实体框架中，需要你向构造函数提供提供程序固定名称。提供程序固定名称用于标识要使用的 SQL Compact 的版本。默认情况下，下面的配置将导致上下文使用 SQL Compact 4.0 版。

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="System.Data.SqlClient.4.0" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

如果未设置默认的连接工厂，则 Code First 使用`SqlConnectionFactory`，指向 `.\SQLEXPRESS`。

`SqlConnectionFactory` 还具有一个构造函数，该构造函数允许你重写连接字符串的各个部分。如果要使用 `.\SQLEXPRESS` 之外的 SQL Server 实例，则可以使用此构造函数来设置服务器。

以下配置将导致 Code First 为未设置显式连接字符串的上下文使用`MyDatabaseServer`。

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">
    <parameters>
      <parameter value="Data Source=MyDatabaseServer; Integrated Security=True;
MultipleActiveResultSets=True" />
    </parameters>
  </defaultConnectionFactory>
</entityFramework>
```

默认情况下，假定构造函数参数的类型为字符串。你可以使用`type`特性来更改此。

```
<parameter value="2" type="System.Int32" />
```

数据库初始值设定项

数据库初始值设定项在每个上下文基础上进行配置。可以使用`上下文`元素在配置文件中设置它们。此元素使用程序集限定名称标识要配置的上下文。

默认情况下，Code First 上下文配置为使用`CreateDatabaseIfNotExists` 初始值设定项。`上下文`元素上有一个可用于禁用数据库初始化的`disableDatabaseInitialization`属性。

例如，下面的配置将禁用在`MyAssembly` 中定义的`BlogContext` 上下文的数据库初始化。

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly" disableDatabaseInitialization="true" />
</contexts>
```

可以使用**databaseInitializer**元素设置自定义初始值设定项。

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly" />
  </context>
</contexts>
```

构造函数参数与默认连接工厂使用相同的语法。

```
<contexts>
  <context type=" Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly">
      <parameters>
        <parameter value="MyConstructorParameter" />
      </parameters>
    </databaseInitializer>
  </context>
</contexts>
```

可以配置实体框架中包含的其中一个泛型数据库初始值设定项。**Type**特性使用泛型类型的 .NET Framework 格式。

例如，如果使用 Code First 迁移，则可以将数据库配置为使用

`MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>` 初始值设定项自动迁移。

```
<contexts>
  <context type="Blogging.BlogContext, MyAssembly">
    <databaseInitializer type="System.Data.Entity.Migrations.DatabaseInitializer`2[[Blogging.BlogContext,
MyAssembly], [Blogging.Migrations.Configuration, MyAssembly]], EntityFramework" />
  </context>
</contexts>
```

连接字符串和模型

2020/3/11 •

本主题介绍实体框架如何发现要使用的数据库连接，以及如何对其进行更改。本主题介绍了 Code First 和 EF 设计器创建的模型。

通常实体框架的应用程序使用派生自 DbContext 的类。此派生类将调用基 DbContext 类上的构造函数之一来控制：

- 上下文将如何连接到数据库-即，如何查找或使用连接字符串
- 上下文是使用 Code First 计算模型还是加载使用 EF 设计器创建的模型
- 其他高级选项

以下片段显示了可使用 DbContext 构造函数的一些方式。

将 Code First 与按约定连接

如果你未在应用程序中进行任何其他配置，则在 DbContext 上调用无参数构造函数将导致 DbContext 在 Code First 模式下运行，并使用约定创建的数据库连接。例如：

```
namespace Demo.EF
{
    public class BloggingContext : DbContext
    {
        public BloggingContext()
        // C# will call base class parameterless constructor by default
        {
        }
    }
}
```

在此示例中，DbContext 使用派生上下文类的命名空间限定名称 ("Bloggingcontext") 作为数据库名称，并使用 SQL Express 或 LocalDB 创建此数据库的连接字符串。如果两者都已安装，将使用 SQL Express。

Visual Studio 2010 默认情况下包括 SQL Express 和 Visual Studio 2012 及更高版本。在安装过程中，EntityFramework NuGet 包检查哪些数据库服务器可用。然后，NuGet 包将通过设置 Code First 在按约定创建连接时使用的默认数据库服务器来更新配置文件。如果 SQL Express 正在运行，将使用它。如果 SQL Express 不可用，则将改为将 LocalDB 注册为默认值。如果配置文件已包含默认连接工厂的设置，则不会对其进行任何更改。

使用具有按约定和指定数据库名称连接的 Code First

如果你未在应用程序中进行任何其他配置，则使用你要使用的数据库名称在 DbContext 上调用字符串构造函数将导致 DbContext 在 Code First 模式下运行，并将数据库连接通过约定创建到数据库该名称。例如：

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingDatabase")
    {
    }
}
```

在此示例中，DbContext 使用 "BloggingDatabase" 作为数据库名称，并使用 SQL Express（与 Visual Studio 2010 —

起安装)或 LocalDB (随 Visual Studio 2012 一起安装)为此数据库创建连接字符串。如果两者都已安装, 将使用 SQL Express。

结合 app.config/web.config 文件中的连接字符串使用 Code First

你可以选择将连接字符串放在 app.config 或 web.config 文件中。例如:

```
<configuration>
  <connectionStrings>
    <add name="BloggingCompactDatabase"
      providerName="System.Data.SqlServerCe.4.0"
      connectionString="Data Source=Blogging.sdf"/>
  </connectionStrings>
</configuration>
```

这是一种简单的方法来告诉 DbContext 使用 SQL Express 或 LocalDB 之外的数据库服务器-上面的示例指定了一个 SQL Server Compact 版本的数据库。

如果连接字符串的名称与你的上下文名称匹配(无论是否具有命名空间限定), 都将在使用无参数构造函数时通过 DbContext 找到该名称。如果连接字符串名称不同于上下文的名称, 则可以通过将连接字符串名称传递给 DbContext 构造函数, 告诉 DbContext 在 Code First 模式下使用此连接。例如:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("BloggingCompactDatabase")
    {
    }
}
```

或者, 您可以为传递到 DbContext 构造函数的字符串使用窗体 "name =<连接字符串名称>"。例如:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingCompactDatabase")
    {
    }
}
```

此窗体使你希望在配置文件中找到连接字符串。如果找不到具有给定名称的连接字符串, 则会引发异常。

App.config/web.config 文件中包含连接字符串的数据库/Model First

用 EF 设计器创建的模型与 Code First 不同, 因为模型已存在, 并且不是在应用程序运行时从代码生成的。模型通常作为 EDMX 文件存在于你的项目中。

设计器会将 EF 连接字符串添加到 app.config 或 web.config 文件。此连接字符串非常特殊, 其中包含了有关如何查找 EDMX 文件中的信息的信息。例如:

```
<configuration>
  <connectionStrings>
    <add name="Northwind_Entities"
      connectionString="metadata=res://*/Northwind.csdl|
                        res://*/Northwind.ssdl|
                        res://*/Northwind.msl;
      provider=System.Data.SqlClient;
      provider connection string=
        "Data Source=.\sqlexpress;
          Initial Catalog=Northwind;
          Integrated Security=True;
          MultipleActiveResultSets=True";
      providerName="System.Data.EntityClient"/>
  </connectionStrings>
</configuration>
```

EF 设计器还将生成代码，该代码通过将连接字符串名称传递到 `DbContext` 构造函数来告诉 `DbContext` 使用此连接。例如：

```
public class NorthwindContext : DbContext
{
    public NorthwindContext()
        : base("name=Northwind_Entities")
    {
    }
}
```

`DbContext` 知道要加载现有模型（而不是使用 Code First 从代码中计算），因为连接字符串是一个 EF 连接字符串，其中包含要使用的模型的详细信息。

其他 `DbContext` 构造函数选项

`DbContext` 类包含其他构造函数和使用模式，可实现更高级的方案。其中一些是：

- 你可以使用 `DbModelBuilder` 类来生成 Code First 模型，而无需实例化 `DbContext` 实例。这是一个 `DbModel` 对象。然后，在准备好创建 `DbContext` 实例时，可以将此 `DbModel` 对象传递给 `DbContext` 构造函数之一。
- 可以将完整的连接字符串传递给 `DbContext`，而不只是数据库或连接字符串的名称。默认情况下，此连接字符串与 `SqlClient` 提供程序一起使用。可以通过将 `IConnectionFactory` 的不同实现设置到上下文来更改此设置。`DefaultConnectionFactory`。
- 可以通过将现有的 `DbConnection` 对象传递给 `DbContext` 构造函数来使用该对象。如果连接对象是 `EntityConnection` 的实例，则将使用连接中指定的模型，而不是使用 Code First 来计算模型。如果对象是某个其他类型（例如 `SqlConnection`）的实例，则上下文会将其用于 Code First 模式。
- 可以将现有 `ObjectContext` 传递到 `DbContext` 构造函数，以创建 `DbContext` 包装现有上下文的功能。这可用于使用 `ObjectContext` 但想要在应用程序的某些部分中利用 `DbContext` 的现有应用程序。

依赖项解析

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

从 EF6 开始，实体框架包含用于获取所需服务实现的通用机制。也就是说，当 EF 使用某些接口或基类的实例时，它将要求使用接口或基类的具体实现。这是通过使用 IDbDependencyResolver 接口实现的：

```
public interface IDbDependencyResolver
{
    object GetService(Type type, object key);
}
```

GetService 方法通常由 EF 调用，由 EF 或应用程序提供的 IDbDependencyResolver 的实现进行处理。调用时，类型参数是所请求服务的接口或基类类型，并且密钥对象为 null 或提供有关所请求服务的上下文信息的对象。

除非另有说明，否则返回的任何对象都必须是线程安全的，因为它可用作单一实例。在许多情况下，返回的对象是一个工厂，在这种情况下，工厂本身必须是线程安全的，但从工厂返回的对象不需要是线程安全的，因为每次使用都从工厂请求了一个新实例。

本文不包含有关如何实现 IDbDependencyResolver 的完整详细信息，而是充当 EF 为其调用 GetService 的服务类型（即接口和基类类型）的参考和其中每个对象的密钥对象的语义拨号。

IDatabaseInitializer < TContext >

引入的版本：ef 6.0.0

返回的对象：给定上下文类型的数据库初始值设定项

键：未使用；将为 null

MigrationSqlGenerator> 的函数的 Func <

引入的版本：ef 6.0.0

返回的对象：用于创建 SQL 生成器的工厂，该生成器可用于迁移和导致创建数据库的其他操作（如使用数据库初始值设定项创建数据库）。

Key：包含 ADO.NET 提供程序固定名称的字符串，该名称指定要为其生成 SQL 的数据库的类型。例如，对于密钥“SqlClient”，将返回 SQL Server SQL 生成器。

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息，请参阅[EF6 提供程序模型](#)部分。

Dbproviderservices.createdatabase 的数据的实体。

引入的版本：ef 6.0.0

返回的对象: 用于给定提供程序固定名称的 EF 提供程序

Key: 包含 ADO.NET 提供程序固定名称的字符串, 用于指定需要提供程序的数据库类型。例如, 对于密钥 "SqlClient", 将返回 SQL Server 提供程序。

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

"**IDbConnectionFactory**"。

引入的版本: ef 6.0。0

返回的对象: 当 EF 按约定创建数据库连接时将使用的连接工厂。也就是说, 如果不向 EF 提供连接或连接字符串, 并且在 app.config 或 web.config 中找不到连接字符串, 则将使用此服务按约定创建连接。默认情况下, 更改连接工厂可以允许 EF 使用其他类型的数据库(例如, SQL Server Compact Edition)。

键: 未使用; 将为 null

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

"**IManifestTokenService**"。

引入的版本: ef 6.0。0

返回的对象: 一种服务, 可从连接生成提供程序清单标记。此服务通常以两种方式使用。首先, 可以使用它来避免 Code First 在生成模型时连接到数据库。其次, 它可以用于强制 Code First 为特定的数据库版本生成模型-例如, 即使使用 SQL Server 2008, 也可以强制对 SQL Server 2005 的模型进行强制。

对象生存期: 单一实例--相同的对象可以多次使用, 并且可以同时由不同的线程使用

键: 未使用; 将为 null

"**IDbProviderFactoryService**"。

引入的版本: ef 6.0。0

返回的对象: 可以从给定连接获得提供程序工厂的服务。在 .NET 4.5 中, 提供程序可以从连接中公开访问。在 .NET 4 中, 此服务的默认实现使用一些试探法来查找匹配的提供程序。如果这些失败, 则可以注册此服务的新实现以提供适当的解决方法。

键: 未使用; 将为 null

Func < DbContext, IDbModelCacheKey 的>

引入的版本: ef 6.0。0

返回的对象: 将为给定上下文生成模型缓存键的工厂。默认情况下, EF 按每个提供程序的 DbContext 类型缓存一个模型。此服务的不同实现可用于向缓存键添加其他信息, 如架构名称。

键: 未使用; 将为 null

DbSpatialServices..。

引入的版本: ef 6.0。0

返回的对象: 一个 EF 空间提供程序, 它添加对 geography 和 geometry 空间类型的基本 EF 提供程序的支持。

键: 以两种方式请求 DbSpatialServices。首先, 使用 DbProviderInfo 对象(其中包含固定名称和清单标记)来请求特定于提供程序的空间服务作为密钥。其次, 无关键字就可以要求 DbSpatialServices。这用于解析在创建独立的 DbGeography 或 DbGeometry 类型时使用的 "全局空间提供程序"。

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

函数函数的 Func < > IDbExecutionStrategy

引入的版本: ef 6.0。0

返回的对象: 一个工厂, 用于创建一个服务, 该服务允许提供程序在针对数据库执行查询和命令时实现重试或其他行为。如果未提供实现, 则 EF 只会执行命令并传播引发的任何异常。对于 SQL Server 此服务用于提供重试策略, 这在对基于云的数据库服务器(如 SQL Azure)运行时特别有用。

Key: 包含提供程序固定名称的 ExecutionStrategyKey 对象, 还可以选择要对其使用执行策略的服务器名称。

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

Func < DbConnection, string, HistoryContext, >

引入的版本: ef 6.0。0

返回的对象: 允许提供程序配置 HISTORYCONTEXT 与 EF 迁移使用的 `_MigrationHistory` 表的映射的工厂。

HistoryContext 是一个 Code First DbContext, 可以使用普通 Fluent API 进行配置, 以更改表名称和列映射规范等项。

键: 未使用; 将为 null

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

DbProviderFactory。

引入的版本: ef 6.0。0

返回的对象: 要用于给定提供程序固定名称的 ADO.NET 提供程序。

Key: 包含 ADO.NET 提供程序固定名称的字符串

NOTE

此服务通常不会直接更改, 因为默认实现使用普通的 ADO.NET 提供程序注册。有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

"`IProviderInvariantName`"。

引入的版本: ef 6.0。0

返回的对象: 用于确定给定类型的 DbProviderFactory 的提供程序固定名称的服务。此服务的默认实现使用 ADO.NET 提供程序注册。这意味着, 如果 ADO.NET 提供程序未以正常方式注册, 因为 DbProviderFactory 正在由 EF 解析, 则还需要解析此服务。

Key: 需要固定名称的 DbProviderFactory 实例。

NOTE

有关 EF6 中与提供程序相关的服务的更多详细信息, 请参阅[EF6 提供程序模型](#)部分。

ViewGeneration. IViewAssemblyCache 的数据。

引入的版本: ef 6.0。0

返回的对象: 包含预生成的视图的程序集的缓存。替换通常用于让 EF 知道哪些程序集包含预生成的视图, 而无需执行任何发现。

键: 未使用; 将为 null

复数形式... IPluralizationService

引入的版本: ef 6.0。0

返回的对象: 由 EF 用于复数形式和确定所名称的服务。默认情况下, 使用英语复数形式服务。

键: 未使用; 将为 null

IDbInterceptor 的数据的元数据

引入的版本: ef 6.0。0

返回的对象: 在应用程序启动时应注册的所有侦听器。请注意, 这些对象是使用 GetServices 调用请求的, 所有依赖关系解析程序返回的所有拦截器都将进行注册。

键: 未使用; 将为 null。

Func < DbContext, Action < string>, DatabaseLogFormatter> 的操作中执行的操作

引入的版本: ef 6.0。0

返回的对象: 一个工厂, 用于创建将在上下文时使用的数据库日志格式化程序。在给定的上下文中设置了数据库日志属性。

键: 未使用; 将为 null。

Func < DbContext>

引入的版本: ef 6.1。0

返回的对象: 当上下文不具有可访问的无参数构造函数时, 将用于创建用于迁移的上下文实例的工厂。

Key: 需要工厂的派生 DbContext 的类型的类型对象。

函数函数 IMetadataAnnotationSerializer> 的 < 函数()

引入的版本: ef 6.1。0

返回的对象: 一个工厂，用于创建用于序列化强类型自定义批注的序列化程序，以便可以将其序列化并 DESTERILIZED 到 XML 中以便在 Code First 迁移中使用。

键: 要序列化或反序列化的批注的名称。

函数函数的 Func <> TransactionHandler

引入的版本: ef 6.1。0

返回的对象: 将用于为事务创建处理程序的工厂，以便在处理提交失败的情况下可以应用特殊处理。

Key: 包含提供程序固定名称的 ExecutionStrategyKey 对象，还可以选择要对其使用事务处理程序的服务器名称。

连接管理

2020/3/11 ·

本页介绍了在将连接传递到上下文和数据库的功能时，实体框架的行为。

将连接传递到上下文

EF5 及更早版本的行为

有两个接受连接的构造函数：

```
public DbContext(DbConnection existingConnection, bool contextOwnsConnection)
public DbContext(DbConnection existingConnection, DbCompiledModel model, bool contextOwnsConnection)
```

可以使用这些方法，但必须解决几个限制：

1. 如果将打开的连接传递到其中的任何一个连接，则在框架第一次尝试使用它时，将引发 `InvalidOperationException`，指出它无法重新打开已打开的连接。
2. `ContextOwnsConnection` 标志解释为在释放上下文时是否应释放基础存储连接。但无论此设置如何，在释放上下文时，存储连接始终关闭。因此，如果有多个具有相同连接的 `DbContext`，则在第一次处理上下文时将关闭连接（同样，如果已使用 `DbContext` 混合了现有的 ADO.NET 连接，`DbContext` 将始终在其释放时关闭连接）。

通过传递关闭的连接并仅执行在创建所有上下文后将打开该连接的代码，可以绕过上述首个限制：

```
using System.Collections.Generic;
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;
using System.Linq;

namespace ConnectionManagementExamples
{
    class ConnectionManagementExampleEF5
    {
        public static void TwoDbContextsOneConnection()
        {
            using (var context1 = new BloggingContext())
            {
                var conn =
                    ((EntityConnection)
                        ((IObjectContextAdapter)context1).ObjectContext.Connection)
                        .StoreConnection;

                using (var context2 = new BloggingContext(conn, contextOwnsConnection: false))
                {
                    context2.Database.ExecuteSqlCommand(
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'");

                    var query = context1.Posts.Where(p => p.Blog.Rating > 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }
                    context1.SaveChanges();
                }
            }
        }
    }
}
```

第二个限制是指您需要避免释放任何 DbContext 对象，直到您已准备好连接才能关闭。

EF6 和未来版本中的行为

在 EF6 和未来的版本中，DbContext 具有相同的两个构造函数，但不再要求在收到时传递到构造函数的连接被关闭。现在可以这样做：

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace ConnectionManagementExamples
{
    class ConnectionManagementExample
    {
        public static void PassingAnOpenConnection()
        {
            using (var conn = new SqlConnection("{connectionString}"))
            {
                conn.Open();

                var sqlCommand = new SqlCommand();
                sqlCommand.Connection = conn;
                sqlCommand.CommandText =
                    @"UPDATE Blogs SET Rating = 5" +
                    " WHERE Name LIKE '%Entity Framework%'";
                sqlCommand.ExecuteNonQuery();

                using (var context = new BloggingContext(conn, contextOwnsConnection: false))
                {
                    var query = context.Posts.Where(p => p.Blog.Rating > 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }
                    context.SaveChanges();
                }

                var sqlCommand2 = new SqlCommand();
                sqlCommand2.Connection = conn;
                sqlCommand2.CommandText =
                    @"UPDATE Blogs SET Rating = 7" +
                    " WHERE Name LIKE '%Entity Framework Rocks%'";
                sqlCommand2.ExecuteNonQuery();
            }
        }
    }
}

```

此外, `contextOwnsConnection` 标志现在控制在释放 `DbContext` 时是否关闭并释放连接。因此, 在上述示例中, 当上下文被释放(行32)时, 连接将不会关闭, 因为它已在 EF 的以前版本中, 而是在断开连接本身的情况下(第40行)。

当然, `DbContext` 仍有可能控制连接(只需将 `contextOwnsConnection` 设置为 `true`, 或者使用其他构造函数之一)。

NOTE

在此新模型中使用事务时, 还有一些其他注意事项。有关详细信息, 请参阅使用[事务](#)。

Connection. Open ()

EF5 及更早版本的行为

在 EF5 及更早版本中, 存在一个 bug, 以便不会更新 `ObjectContext`, 以反映基础存储连接的真实状态。例如, 如果执行了下面的代码, 则可以将状态视为已关闭, 即使基础存储连接已打开。

```
((ObjectContextAdapter)context).ObjectContext.Connection.State
```

另外，如果您通过调用 "数据库连接" 打开数据库连接()，则在下一次执行查询或调用需要数据库连接的任何内容（例如 SaveChanges ()）之后但在底层存储区之后，就会打开数据库连接。连接将关闭。然后，上下文将重新打开并在每次需要另一数据库操作时重新关闭连接：

```
using System;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;

namespace ConnectionManagementExamples
{
    public class DatabaseOpenConnectionBehaviorEF5
    {
        public static void DatabaseOpenConnectionBehavior()
        {
            using (var context = new BloggingContext())
            {
                // At this point the underlying store connection is closed

                context.Database.Connection.Open();

                // Now the underlying store connection is open
                // (though ObjectContext.Connection.State will report closed)

                var blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);

                // The underlying store connection is still open

                context.SaveChanges();

                // After SaveChanges() the underlying store connection is closed
                // Each SaveChanges() / query etc now opens and immediately closes
                // the underlying store connection

                blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();
            }
        }
    }
}
```

EF6 和未来版本中的行为

对于 EF6 和更早的版本，如果调用代码选择通过调用上下文来打开连接，我们采用了这种方法。然后打开()，它有一个很好的理由来完成此操作，框架将假定它要控制连接的打开和关闭，并且将不再自动关闭连接。

NOTE

这可能会导致连接长时间处于打开状态，因此请谨慎使用。

我们还更新了代码，使 ObjectContext 连接状态立即跟踪基础连接的状态。

```
using System;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Core.EntityClient;
using System.Data.Entity.Infrastructure;

namespace ConnectionManagementExamples
{
    internal class DatabaseOpenConnectionBehaviorEF6
    {
        public static void DatabaseOpenConnectionBehavior()
        {
            using (var context = new BloggingContext())
            {
                // At this point the underlying store connection is closed

                context.Database.Connection.Open();

                // Now the underlying store connection is open and the
                // ObjectContext.Connection.State correctly reports open too

                var blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();

                // The underlying store connection remains open for the next operation

                blog = new Blog { /* Blog's properties */ };
                context.Blogs.Add(blog);
                context.SaveChanges();

                // The underlying store connection is still open

            } // The context is disposed - so now the underlying store connection is closed
        }
    }
}
```

连接复原和重试逻辑

2020/3/16 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

由于后端故障和网络不稳定，连接到数据库服务器的应用程序始终容易遭受连接中断。然而，在基于 LAN 的环境中，对于专用数据库服务器，这些错误很少发生，但通常不需要使用额外的逻辑来处理这些故障。随着基于云的数据库服务器（如 Microsoft Azure SQL 数据库）的增长以及通过不太可靠的网络进行的连接，现在更常见的连接中断。这可能是因为云数据库使用防御性技术来确保服务的公平，如连接限制，或导致间歇超时和其他暂时性错误的网络不稳定。

连接复原是指 EF 自动重试任何由于这些连接中断而失败的命令的功能。

执行策略

连接重试由 `IDbExecutionStrategy` 接口的实现来处理。`IDbExecutionStrategy` 的实现将负责接受一个操作，如果发生异常，则确定重试是否合适，如果是，则重试。EF 附带四个执行策略：

1. `DefaultExecutionStrategy`：此执行策略不会重试任何操作，它是 sql server 以外的数据库的默认值。
2. `DefaultSqlExecutionStrategy`：这是默认情况下使用的内部执行策略。不过，这种策略根本不会重试，它将包装任何可能会暂时性的异常，以通知用户他们可能想要启用连接复原。
3. `DbExecutionStrategy`：此类适用于其他执行策略（包括你自己的自定义）的基类。它实现了一个指数重试策略，其中，初始重试会出现零延迟，延迟将以指数方式递增，直到达到最大重试计数。此类有一个抽象 `ShouldRetryOn` 方法，可在派生执行策略中实现该方法，以控制应重试哪些异常。
4. `SqlAzureExecutionStrategy`：此执行策略继承自 `DbExecutionStrategy`，并将在使用 Azure SQL 数据库时已知可能会暂时性的异常重试。

NOTE

执行策略2和4包含在 EF 附带的 Sql Server 提供程序中，后者位于 EntityFramework 程序集中，旨在与 SQL Server 配合使用。

启用执行策略

告诉 EF 使用执行策略的最简单方法是使用[DbConfiguration](#)类的 `SetExecutionStrategy` 方法：

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());
    }
}
```

此代码告知 EF 在连接到 SQL Server 时使用 `SqlAzureExecutionStrategy`。

配置执行策略

`SqlAzureExecutionStrategy` 的构造函数可以接受两个参数：`MaxRetryCount` 和 `MaxDelay`。`MaxRetryCount` 是策略将重试的最大次数。`MaxDelay` 是一个 `TimeSpan`, 表示执行策略将使用的重试之间的最大延迟。

若要将最大重试次数设置为1, 最大延迟为30秒, 请执行以下操作:

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetExecutionStrategy(
            "System.Data.SqlClient",
            () => new SqlAzureExecutionStrategy(1, TimeSpan.FromSeconds(30)));
    }
}
```

第一次发生暂时性故障时, `SqlAzureExecutionStrategy` 会立即重试, 但会在每次重试之间延迟更长的时间, 直到超过最大重试限制或总时间达到最大延迟。

执行策略只会重试有限数量的异常, 这些异常通常是暂时性的, 你仍需要处理其他错误, 并捕获 `RetryLimitExceeded` 异常, 以防错误不是暂时性的, 或者需要太长时间才能解决自动.

使用重试执行策略时, 有一些已知的限制:

不支持流式处理查询

默认情况下, EF6 和更高版本将缓冲查询结果, 而不是流式传输。如果希望流式传输结果, 可以使用 `AsStreaming` 方法将 LINQ to Entities 查询更改为流式处理。

```
using (var db = new BloggingContext())
{
    var query = (from b in db.Blogs
                 orderby b.Url
                 select b).AsStreaming();
}
```

注册重试执行策略时, 不支持流式处理。存在此限制的原因是, 连接可能会在返回的结果中下降。出现这种情况时, EF 需要重新运行整个查询, 但无法可靠地了解已返回的结果(数据自初始查询发送以来可能已更改, 结果可能会以不同的顺序返回, 结果可能不具有唯一标识符, 等等)。

不支持用户启动的事务

配置导致重试的执行策略时, 对事务的使用存在一些限制。

默认情况下, EF 将在事务中执行任何数据库更新。不需要执行任何操作来启用此功能, EF 始终会自动执行此操作。

例如, 在以下代码中, 将在事务中自动执行。如果在插入新的一个站点后, `SaveChanges` 会失败, 则会回滚该事务, 并且不会将更改应用到数据库。上下文还会处于状态, 该状态允许再次调用 `SaveChanges` 以重试应用更改。

```
using (var db = new BloggingContext())
{
    db.Blogs.Add(new Site { Url = "http://msdn.com/data/ef" });
    db.Blogs.Add(new Site { Url = "http://blogs.msdn.com/adonet" });
    db.SaveChanges();
}
```

如果不使用重试执行策略, 则可以在单个事务中包装多个操作。例如, 下面的代码将两个 `SaveChanges` 调用包装

在一个事务中。如果任一操作的任何部分失败，则不会应用任何更改。

```
using (var db = new BloggingContext())
{
    using (var trn = db.Database.BeginTransaction())
    {
        db.Blogs.Add(new Site { Url = "http://msdn.com/data/ef" });
        db.Blogs.Add(new Site { Url = "http://blogs.msdn.com/adonet" });
        db.SaveChanges();

        db.Blogs.Add(new Site { Url = "http://twitter.com/efmagicunicorns" });
        db.SaveChanges();

        trn.Commit();
    }
}
```

使用重试执行策略时不支持此操作，因为 EF 不知道任何以前的操作以及如何重试。例如，如果第二个 SaveChanges 失败，则 EF 不再具有重试第一个 SaveChanges 调用所需的信息。

解决方案：手动调用执行策略

解决方案是手动使用执行策略，并为其分配要运行的整个逻辑集，以便在其中一个操作失败的情况下可以重试所有操作。当派生自 DbExecutionStrategy 的执行策略正在运行时，它将挂起 SaveChanges 中使用的隐式执行策略。

请注意，应在代码块内构造要重试的所有上下文。这可确保每次重试都从干净的状态开始。

```
var executionStrategy = new SqlAzureExecutionStrategy();

executionStrategy.Execute(
    () =>
{
    using (var db = new BloggingContext())
    {
        using (var trn = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "http://msdn.com/data/ef" });
            db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "http://twitter.com/efmagicunicorns" });
            db.SaveChanges();

            trn.Commit();
        }
    }
});
```

处理事务提交失败

2020/3/11 ·

NOTE

EF 6.1 在实体框架 6.1 中引入了本页中讨论的功能、API 等。如果使用的是早期版本，则部分或全部信息不适用。

作为 6.1 的一部分，我们将为 EF 引入新的连接复原功能：当暂时性连接故障影响事务提交确认时，自动检测和恢复。有关该方案的完整详细信息，请参阅博客文章[SQL 数据库连接和幂等性问题](#)。总的来说，方案是在事务提交期间引发异常时，有两个可能的原因：

1. 服务器上的事务提交失败
2. 服务器上的事务提交成功，但连接问题阻止成功通知到达客户端

在第一种情况下，应用程序或用户可以重试该操作，但当发生第二种情况时，应避免重试，并且应用程序可能会自动恢复。难点在于，如果不能检测到在提交期间报告异常的实际原因，应用程序将无法选择正确的操作过程。EF 6.1 中的新功能允许 EF 在事务成功并以透明方式执行正确操作的过程中仔细检查数据库。

使用功能

若要启用此功能，需要在 `DbConfiguration` 的构造函数中包括对 `SetTransactionHandler` 的调用。如果不熟悉 `DbConfiguration`，请参阅[基于代码的配置](#)。此功能可以与我们在 EF6 中引入的自动重试结合使用，这有助于在发生暂时性故障的情况下，事务实际上未能在服务器上提交：

```
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.SqlServer;

public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetTransactionHandler(SqlProviderServices.ProviderInvariantName, () => new CommitFailureHandler());
        SetExecutionStrategy(SqlProviderServices.ProviderInvariantName, () => new SqlAzureExecutionStrategy());
    }
}
```

如何跟踪事务

启用此功能后，EF 会自动将新表添加到名为 `_Transactions` 的数据库。在此表中，每次创建事务时都会在此表中插入一个新行，并在提交期间发生事务失败时检查该行是否存在。

尽管 EF 在不再需要时可以最大程度地修剪表中的行，但如果应用程序过早退出，则表可能会增长，因此，在某些情况下，可能需要手动清除该表。

如何处理以前版本的提交失败

EF 6.1 之前没有处理 EF 产品中的提交失败的机制。可以通过多种方式来处理可应用于早期版本的 EF6 的情况：

- 选项 1-不执行任何操作

事务提交期间连接失败的可能性很低，因此如果实际发生此情况，应用程序可能会失败。

- 选项 2-使用数据库重置状态
 1. 放弃当前 DbContext
 2. 创建新的 DbContext 并从数据库还原应用程序的状态
 3. 通知用户上一次操作可能未成功完成
- 选项 3-手动跟踪事务
 1. 将非跟踪表添加到用于跟踪事务状态的数据库。
 2. 在每个事务开头的表中插入一行。
 3. 如果在提交期间连接失败, 请检查数据库中是否存在相应的行。
 - 如果行存在, 则继续正常, 因为事务已成功提交
 - 如果该行不存在, 请使用执行策略重试当前操作。
 4. 如果提交成功, 则删除相应的行以避免表增长。

[此博客文章](#)包含 SQL Azure 的示例代码。

通过 WinForms 进行数据绑定

2020/3/11 ·

此分步演练演示如何将 POCO 类型绑定到“主/详细信息”窗体中的窗口窗体(WinForms)控件。应用程序使用实体框架使用数据库中的数据填充对象、跟踪更改并将数据保存到数据库。

该模型定义了两种参与一对多关系的类型：类别(主体\主)和产品(依赖\详细信息)。然后，使用 Visual Studio 工具将模型中定义的类型绑定到 WinForms 控件。WinForms 数据绑定框架允许在相关对象之间导航：在主视图中选择行后，详细信息视图将更新为相应的子数据。

本演练中的屏幕截图和代码清单取自 Visual Studio 2013，但你可以通过 Visual Studio 2012 或 Visual Studio 2010 完成此演练。

先决条件

需要安装 Visual Studio 2013、Visual Studio 2012 或 Visual Studio 2010 才能完成此演练。

如果你使用的是 Visual Studio 2010，则还必须安装 NuGet。有关详细信息，请参阅[安装 NuGet](#)。

创建应用程序

- 打开 Visual Studio
- 文件->> 项目。
- 在左侧窗格中选择 “windows”，在右窗格中选择 “windows FormsApplication”
- 输入WinFormswithEFSample作为名称
- 选择“确定”

安装实体框架 NuGet 包

- 在解决方案资源管理器中，右键单击WinFormswithEFSample项目
- 选择 “管理 NuGet 包 ...”
- 在 “管理 NuGet 包” 对话框中，选择 “联机” 选项卡，然后选择 “EntityFramework” 包
- 单击“安装”

NOTE

除了 EntityFramework 程序集之外，还添加了对 System.componentmodel 的引用。如果项目具有对 EntityFramework 的引用，则在安装包时将被删除。System.web 程序集不再用于实体框架6应用程序。

为集合实现 IListSource

当使用 Windows 窗体时，集合属性必须实现 IListSource 接口以启用使用排序的双向数据绑定。为此，我们将扩展 ObservableCollection 以添加 IListSource 功能。

- 将ObservableListSource类添加到项目：
 - 右键单击项目名称
 - 选择 “添加-> 新项”
 - 选择类，并输入ObservableListSource作为类名
- 将默认生成的代码替换为以下代码：

此类启用双向数据绑定和排序。类从 *ObservableCollection* 派生 *<T>* 并添加 *IListSource* 的显式实现。实现 *IListSource* 的执行 *getlist ()* 方法以返回保持与 *ObservableCollection* 同步的 *IBindingList* 实现。对 *tobindinglist* 获得生成的 *IBindingList* 实现支持排序。对 *tobindinglist* 获得扩展方法是在 *EntityFramework* 程序集中定义的。

```
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Diagnostics.CodeAnalysis;
using System.Data.Entity;

namespace WinFormswithEFSample
{
    public class ObservableListSource<T> : ObservableCollection<T>, IListSource
        where T : class
    {
        private IBindingList _bindingList;

        bool IListSource.ContainsListCollection { get { return false; } }

        IList IListSource.GetList()
        {
            return _bindingList ?? (_bindingList = this.TobindingList());
        }
    }
}
```

定义模型

在本演练中，您可以使用 *Code First* 或 *EF* 设计器选择实现模型。完成以下两个部分中的一个。

选项1：使用 *Code First* 定义模型

本部分说明如何使用 *Code First* 创建模型及其关联的数据库。如果要 *Database First* 使用 *EF* 设计器从数据库中反向工程模型，请跳到下一节（[选项2：使用 *Database First* 定义模型](#)）

使用 *Code First* 开发时，通常首先编写定义概念（域）模型 .NET Framework 类。

- 将新产品类添加到项目
- 将默认生成的代码替换为以下代码：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }

        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}
```

- 将*Category*类添加到项目。
- 将默认生成的代码替换为以下代码：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Category
    {
        private readonly ObservableListSource<Product> _products =
            new ObservableListSource<Product>();

        public int CategoryId { get; set; }
        public string Name { get; set; }
        public virtual ObservableListSource<Product> Products { get { return _products; } }
    }
}

```

除了定义实体外，还需要定义派生自`DbContext`的类，并公开`DbSet< TEntity >`属性。`DbSet`属性让上下文知道要包括在模型中的类型。`DbContext`和`DbSet`类型是在 EntityFramework 程序集中定义的。

`DbContext`派生类型的实例在运行时管理实体对象，这包括使用数据库中的数据填充对象、更改跟踪以及将数据保存到数据库。

- 向项目中添加一个新的`ProductContext`类。
- 将默认生成的代码替换为以下代码：

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;

namespace WinFormswithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}

```

编译该项目。

选项2：使用 Database First 定义模型

本部分说明如何使用 Database First 从使用 EF 设计器的数据库中对模型进行反向工程。如果已完成上一部分（选项1：使用 Code First 定义模型），则跳过此部分，直接转到延迟加载部分。

创建现有数据库

通常，当目标为现有数据库时，它将被创建，但在本演练中，我们需要创建一个要访问的数据库。

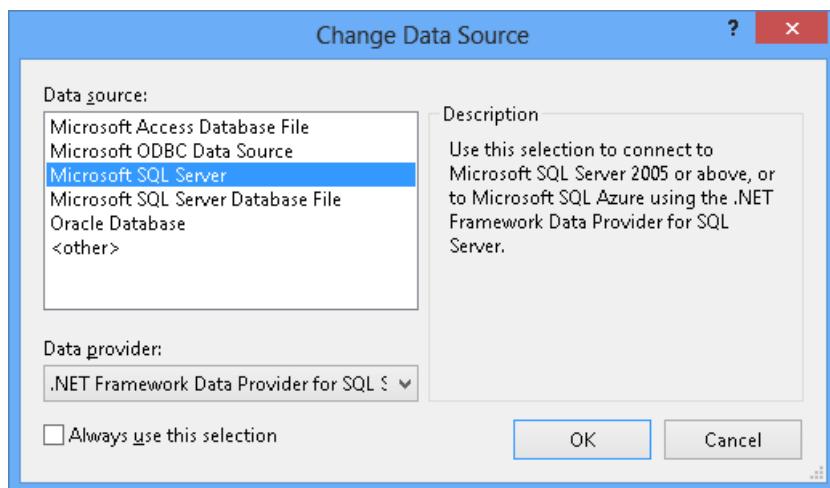
随 Visual Studio 一起安装的数据库服务器因安装的 Visual Studio 版本而异：

- 如果使用的是 Visual Studio 2010，则将创建 SQL Express 数据库。
- 如果使用的是 Visual Studio 2012，则将创建一个[LocalDB](#)数据库。

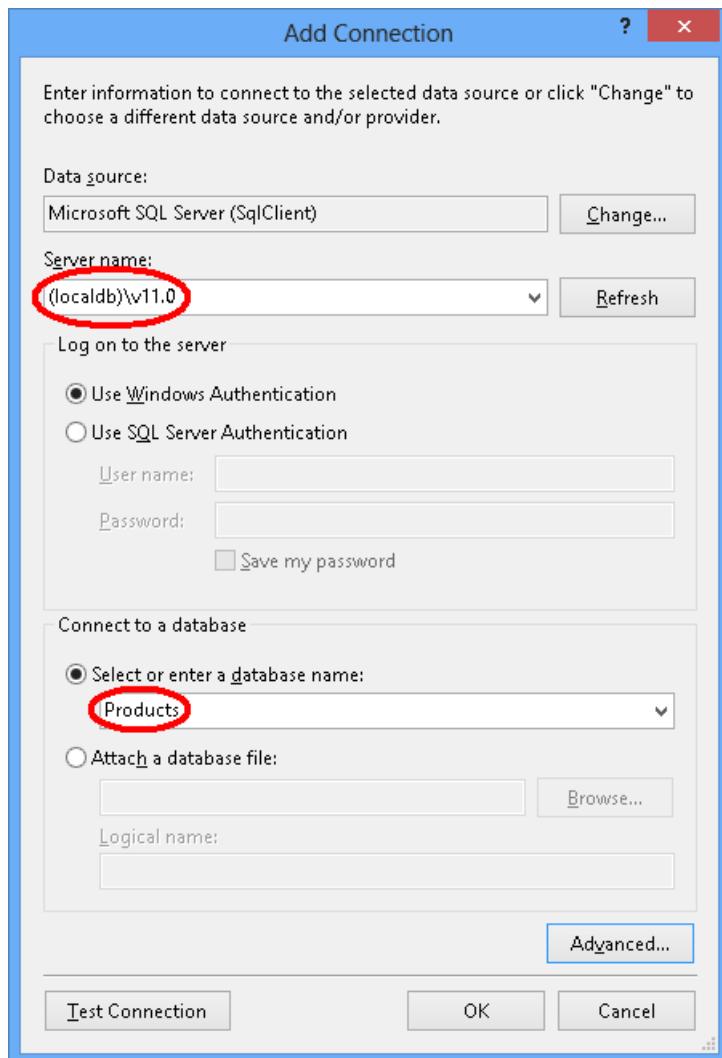
接下来，生成数据库。

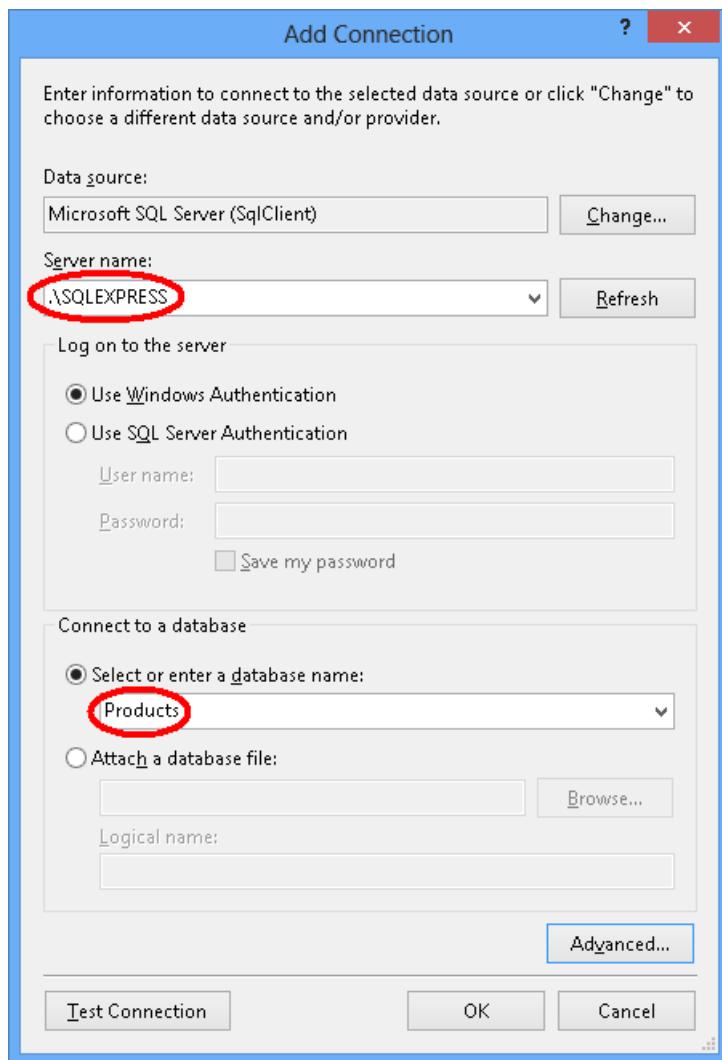
- 视图-> 服务器资源管理器

- 右键单击 "数据连接-> 添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择 Microsoft SQL Server 作为数据源

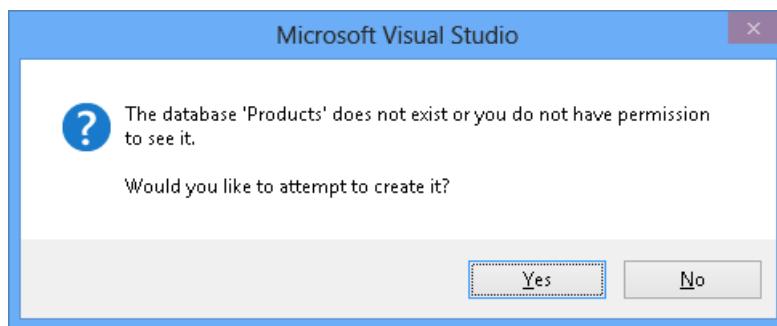


- 连接到 LocalDB 或 SQL Express，具体取决于已安装的数据，并输入Products作为数据库名称





- 选择 "确定"，系统会询问您是否要创建新数据库，请选择 "是"



- 新数据库现在将出现在服务器资源管理器中，右键单击该数据库并选择 "新建查询"
- 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择 "执行"

```

CREATE TABLE [dbo].[Categories] (
    [CategoryId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    CONSTRAINT [PK_dbo.Categories] PRIMARY KEY ([CategoryId])
)

CREATE TABLE [dbo].[Products] (
    [ProductId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    [CategoryId] [int] NOT NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY ([ProductId])
)

CREATE INDEX [IX_CategoryId] ON [dbo].[Products]([CategoryId])

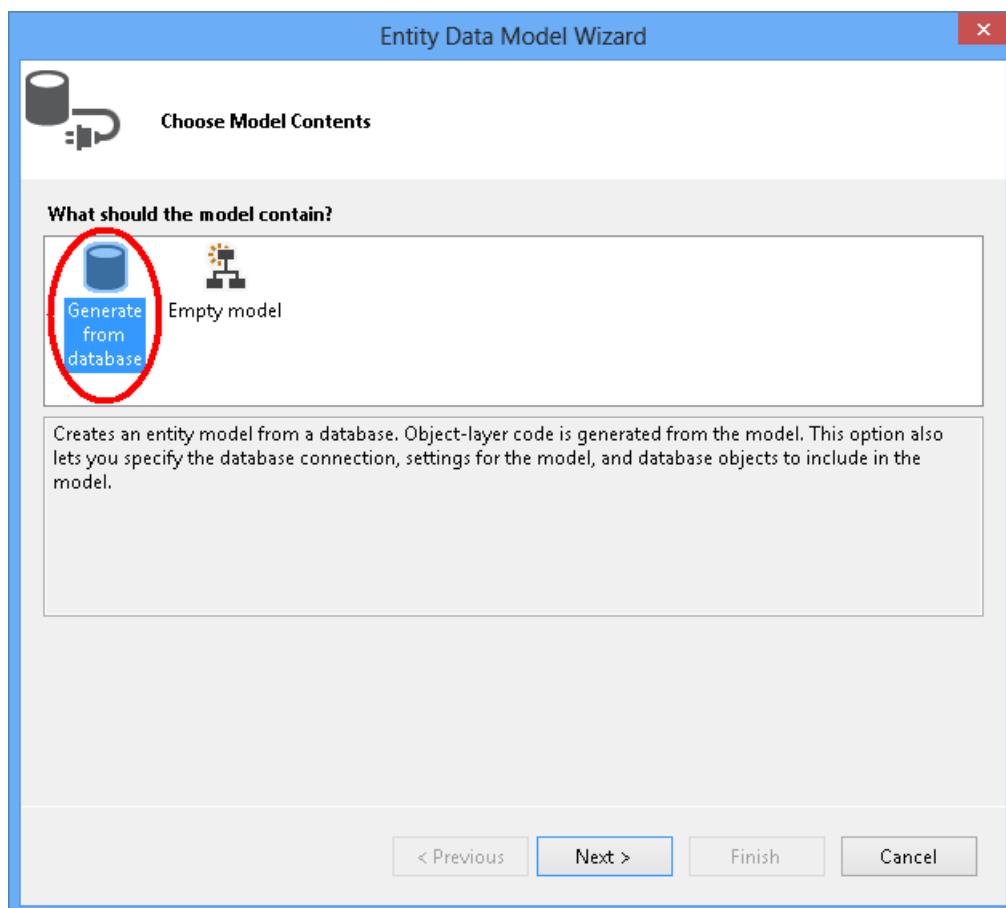
ALTER TABLE [dbo].[Products] ADD CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY
([CategoryId]) REFERENCES [dbo].[Categories] ([CategoryId]) ON DELETE CASCADE

```

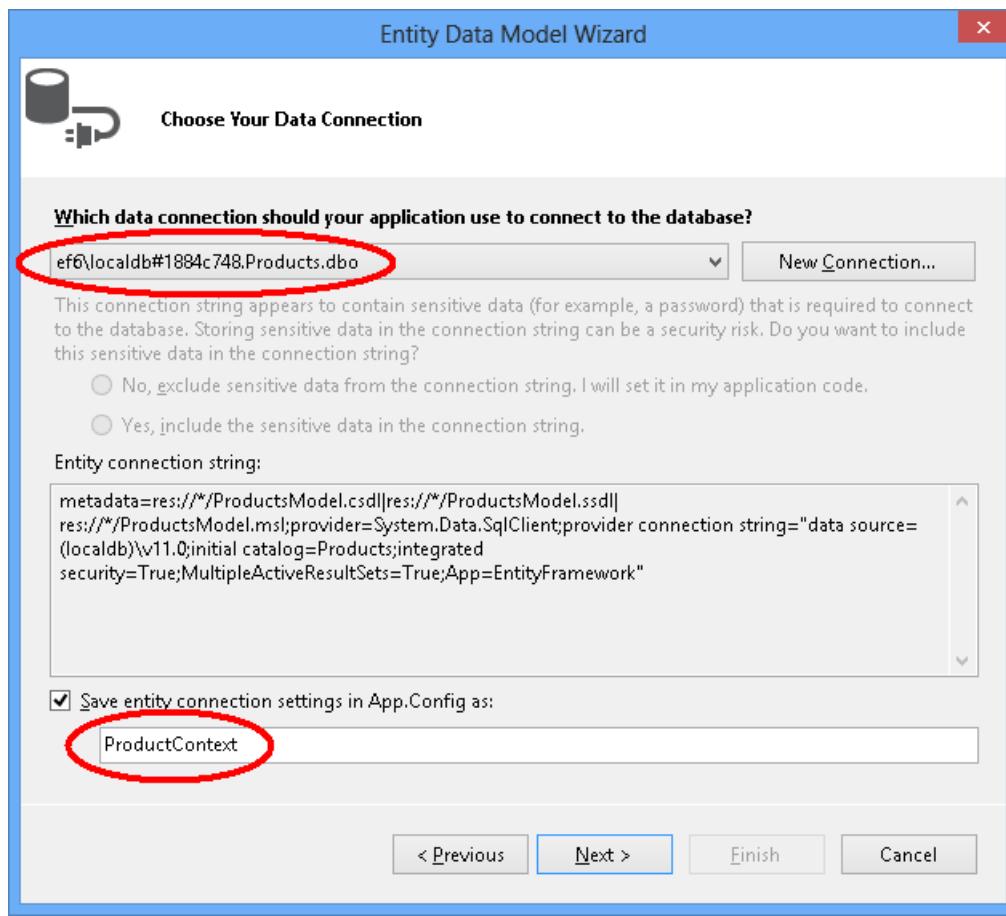
反向工程模型

我们将使用在 Visual Studio 中包含的 Entity Framework Designer 来创建模型。

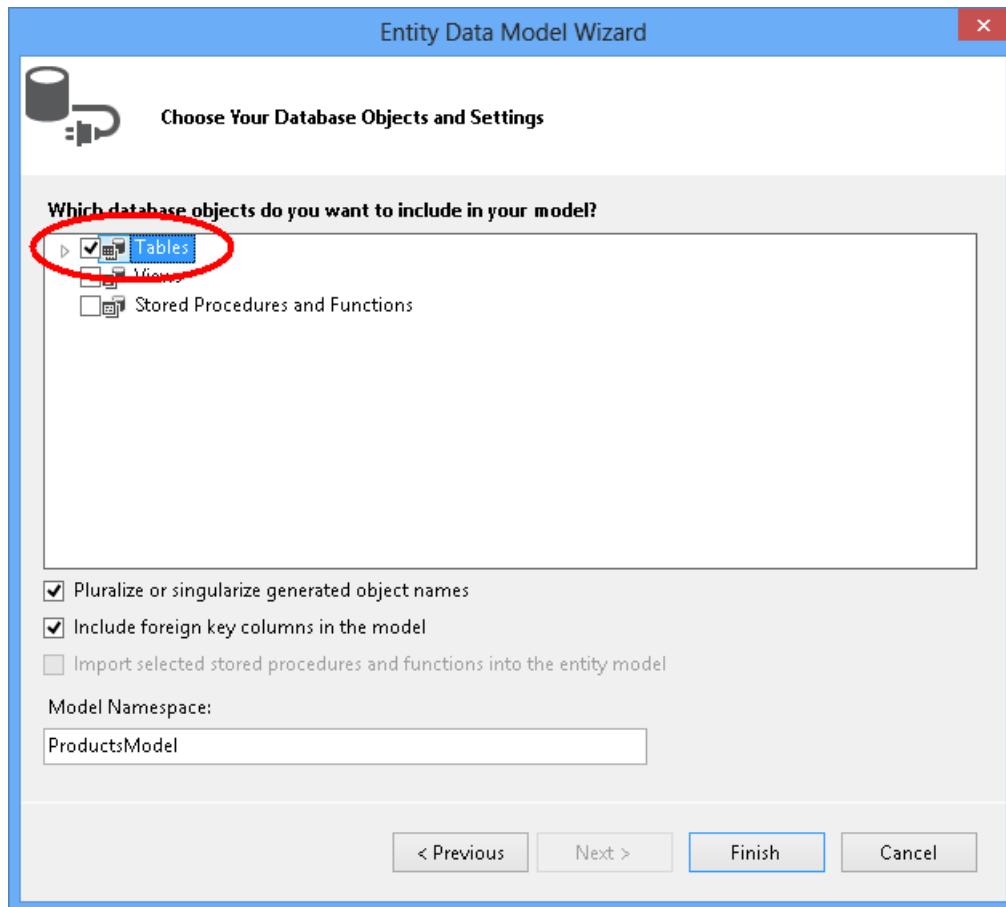
- 项目->“添加新项...”
- 从左侧菜单中选择“数据”，然后ADO.NET 实体数据模型
- 输入ProductModel作为名称，然后单击“确定”
- 这将启动实体数据模型向导
- 选择“从数据库生成”，然后单击“下一步”



- 选择在第一部分中创建的数据库的连接，输入“ProductContext”作为连接字符串的名称，然后单击“下一步”



- 单击 "表" 旁边的复选框以导入所有表, 然后单击 "完成"



反向工程过程完成后, 会将新模型添加到项目中, 并打开, 以便在 Entity Framework Designer 中查看。App.config 文件也已添加到您的项目中, 其中包含数据库的连接详细信息。

Visual Studio 2010 中的其他步骤

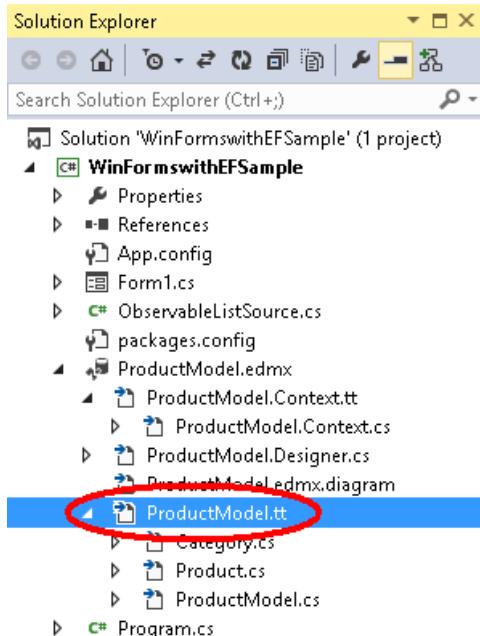
如果使用的是 Visual Studio 2010，则需要更新 EF 设计器以使用 EF6 代码生成。

- 在 EF 设计器中右键单击模型的空位置，然后选择“添加代码生成项...”
- 从左侧菜单中选择“联机模板”，然后搜索DbContext
- 选择用于 C# 的 EF 1.X DbContext 生成器，输入 ProductsModel 作为名称，然后单击“添加”

更新数据绑定的代码生成

EF 使用 T4 模板从模型生成代码。Visual Studio 附带的模板或从 Visual Studio 库下载的模板专用于一般用途。这意味着从这些模板生成的实体具有简单的 ICollection<T> 属性。但是，在执行数据绑定时，需要具有实现 IListSource 的集合属性。这就是我们创建上述 ObservableListSource 类的原因，现在我们要修改模板以使用此类。

- 打开解决方案资源管理器并查找 ProductModel 文件
- 查找 ProductModel.tt 文件，该文件将嵌套在 ProductModel 文件夹下



- 双击 ProductModel.tt 文件以在 Visual Studio 编辑器中将其打开
- 查找并将 "ICollection" 的两个匹配项替换为 "ObservableListSource"。这些代码位于大约第 296 行和第 484 行。
- 查找并将 "HashSet" 的第一个匹配项替换为 "ObservableListSource"。此事件大约位于第 50 行。请勿替换稍后在代码中找到的第二个 HashSet。
- 保存 ProductModel.tt 文件。这会导致重新生成实体的代码。如果代码未自动重新生成，则右键单击 ProductModel.tt 并选择“运行自定义工具”。

如果你现在打开 Category.cs 文件（该文件嵌套在 ProductModel.tt 下），则应看到 Products 集合具有类型 ObservableListSource<产品>。

编译该项目。

延迟加载

Product 类上“类别类”和“类别”属性的“产品”属性是导航属性。在实体框架中，导航属性提供了一种方法，用于在两个实体类型之间导航关系。

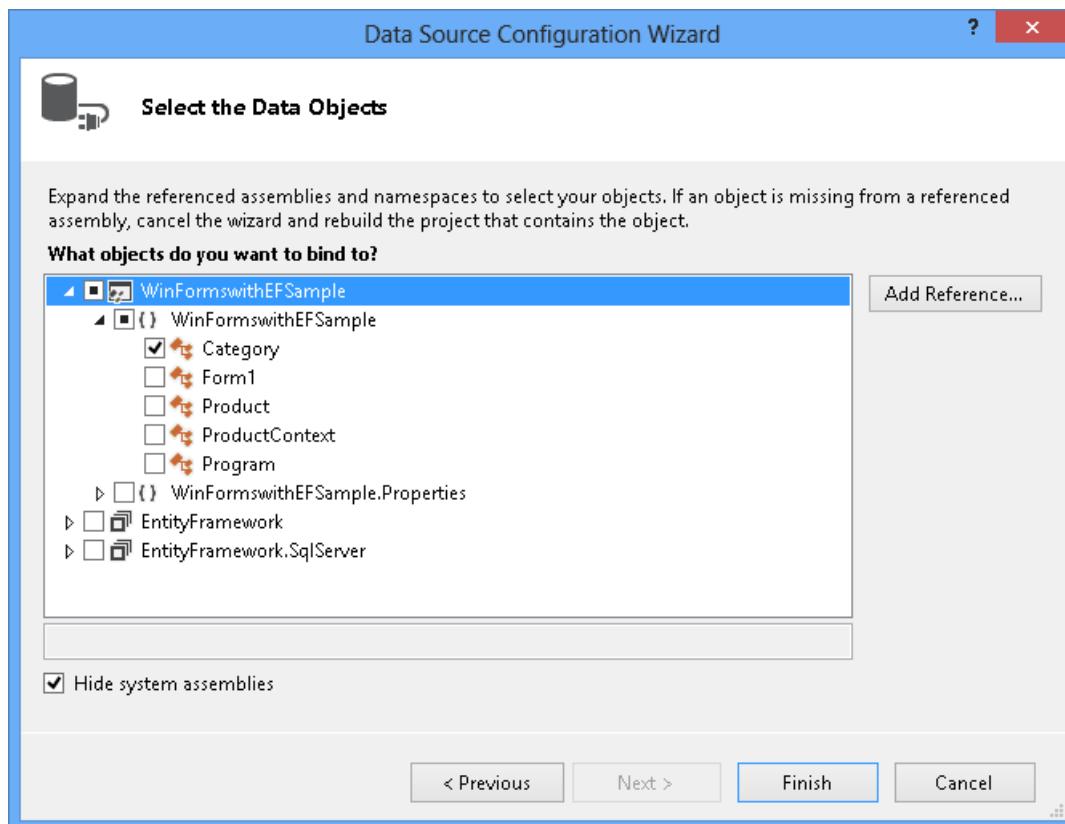
当首次访问导航属性时，EF 使你可以选择自动从数据库加载相关实体。使用这种类型的加载（称为“延迟加载”）时，请注意，第一次访问每个导航属性时，将对数据库执行单独的查询（如果内容尚未出现在上下文中）。

使用 POCO 实体类型时，EF 通过在运行时创建派生代理类型的实例，然后重写类中的虚拟属性来添加加载挂钩来实现延迟加载。若要获取相关对象的延迟加载，必须将导航属性 getter 声明为公共和虚拟（在 Visual Basic 中可重写），并且不能密封类（Visual Basic NotOverridable）。使用时，会自动将 Database First 导航属性设为虚拟，以后用延迟加载。在 "Code First" 部分中，我们选择将导航属性设置为虚拟，因为同一原因

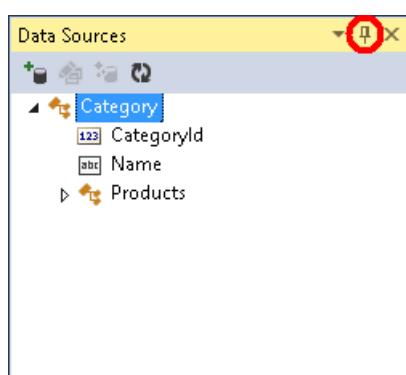
将对象绑定到控件

将在模型中定义的类添加为此 WinForms 应用程序的数据源。

- 从主菜单中，选择 "项目-> 添加新数据源 ..."（在 Visual Studio 2010 中，需要选择 "数据>" 添加新数据源 ...）
- 在 "选择数据源类型" 窗口中，选择 "对象"，然后单击 "下一步"
- 在 "选择数据对象" 对话框中，展开WinFormswithEFSample两次，然后选择 "类别"，因为我们将通过类别数据源上的产品属性来获取产品数据源。



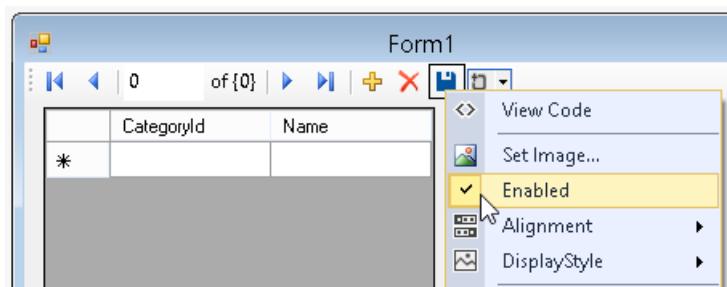
- 单击 "完成"。如果 "数据源" 窗口未显示，请选择 "查看"-> 其他 Windows> 数据源
- 按固定图标，使 "数据源" 窗口不会自动隐藏。如果窗口已显示，可能需要按 "刷新" 按钮。



- 在解决方案资源管理器中，双击Form1.cs文件以在设计器中打开主窗体。
- 选择类别数据源并将其拖到窗体上。默认情况下，新的 DataGridView (categoryDataGridView) 和导航工

具栏控件将添加到设计器中。这些控件绑定到创建的 BindingSource (categoryBindingSource) 和绑定导航器 (categoryBindingNavigator) 组件。

- 编辑 categoryDataGridView 上的列。我们想要将 "类别 id" 列设置为只读。保存数据后，数据库将生成 "类别 id" 属性的值。
 - 右键单击 DataGridView 控件并选择 "编辑列"。
 - 选择 "类别 Id" 列并将 "ReadOnly" 设置为 True
 - 按 "确定"
 - 从类别数据源下选择 "产品"，并将其拖到窗体上。ProductDataGridView 和为 productbindingsource 将添加到窗体中。
 - 编辑 productDataGridView 上的列。我们想要隐藏类别 Id 和类别列，并将 ProductId 设置为只读。在保存数据后，数据库将生成 ProductId 属性的值。
 - 右键单击 DataGridView 控件，然后选择 "编辑列 ..."。
 - 选择 "ProductId" 列并将 "ReadOnly" 设置为 True。
 - 选择 "类别 id" 列并按 "删除" 按钮。对 Category 列执行相同的操作。
 - 按 "确定"。
- 到目前为止，我们在设计器中将 DataGridView 控件与 BindingSource 组件相关联。在下一部分中，我们将向隐藏代码中添加代码，以便将 categoryBindingSource 设置为 DbContext 当前跟踪的实体集合。当我们从类别中拖放产品时，WinForms 会将 productsBindingSource 属性设置为 "categoryBindingSource" 和 "productsBindingSource" 属性设置为 "产品"。由于此绑定，只会在 productDataGridView 中显示属于当前选定类别的产品。
- 通过单击鼠标右键并选择 "启用"，启用导航工具栏上的 "保存" 按钮。



- 双击按钮，为 "保存" 按钮添加事件处理程序。这将添加事件处理程序，并将你带入窗体的隐藏代码。下一部分将添加 categoryBindingNavigatorSaveItem_Click 事件处理程序的代码。

添加处理数据交互的代码

现在，我们将添加代码以使用 ProductContext 来执行数据访问。更新主窗体窗口的代码，如下所示。

该代码声明了 ProductContext 的长时间运行的实例。ProductContext 对象用于查询数据并将数据保存到数据库。然后，从重写的 OnClosing 方法中调用 ProductContext 实例上的 Dispose () 方法。代码注释提供有关代码的作用的详细信息。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.Entity;
```

```

namespace WinFormswithEFSample
{
    public partial class Form1 : Form
    {
        ProductContext _context;
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnLoad(EventArgs e)
        {
            base.OnLoad(e);
            _context = new ProductContext();

            // Call the Load method to get the data for the given DbSet
            // from the database.
            // The data is materialized as entities. The entities are managed by
            // the DbContext instance.
            _context.Categories.Load();

            // Bind the categoryBindingSource.DataSource to
            // all the Unchanged, Modified and Added Category objects that
            // are currently tracked by the DbContext.
            // Note that we need to call ToBindingList() on the
            // ObservableCollection< TEntity > returned by
            // the DbSet.Local property to get the BindingList< T >
            // in order to facilitate two-way binding in WinForms.
            this.categoryBindingSource.DataSource =
                _context.Categories.Local.ToBindingList();
        }

        private void categoryBindingNavigatorSaveItem_Click(object sender, EventArgs e)
        {
            this.Validate();

            // Currently, the Entity Framework doesn't mark the entities
            // that are removed from a navigation property (in our example the Products)
            // as deleted in the context.
            // The following code uses LINQ to Objects against the Local collection
            // to find all products and marks any that do not have
            // a Category reference as deleted.
            // The ToList call is required because otherwise
            // the collection will be modified
            // by the Remove call while it is being enumerated.
            // In most other situations you can do LINQ to Objects directly
            // against the Local property without using ToList first.
            foreach (var product in _context.Products.Local.ToList())
            {
                if (product.Category == null)
                {
                    _context.Products.Remove(product);
                }
            }

            // Save the changes to the database.
            this._context.SaveChanges();

            // Refresh the controls to show the values
            // that were generated by the database.
            this.categoryDataGridView.Refresh();
            this.productsDataGridView.Refresh();
        }

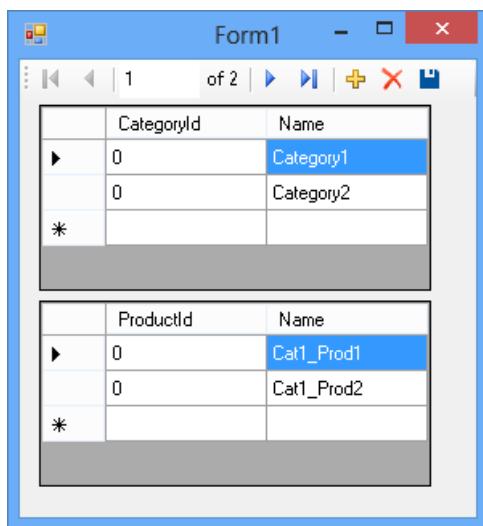
        protected override void OnClosing(CancelEventArgs e)
        {
            base.OnClosing(e);
            this._context.Dispose();
        }
    }
}

```

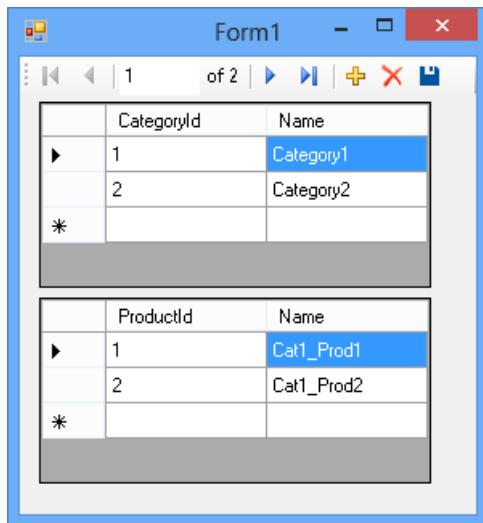
```
    }  
}
```

测试 Windows 窗体应用程序

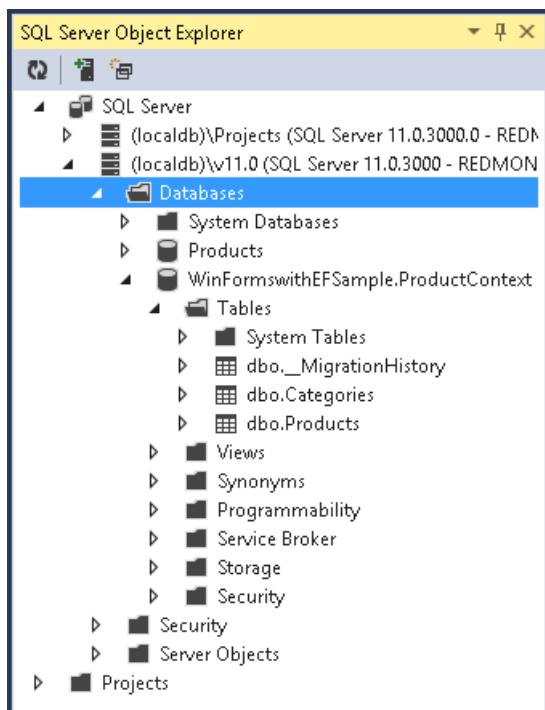
- 编译并运行应用程序，可以测试功能。



- 保存存储生成的密钥后，会显示在屏幕上。



- 如果使用 Code First，则还会看到为你创建WinFormswithEFSample. ProductContext数据库。



IMPORTANT

.NET WPF

本文档介绍 .NET 框架上 WPF 的数据绑定。对于新的 .NET 核心项目，我们建议您使用[EF Core](#)而不是实体框架 6。EF Core 中的数据绑定文档在[“问题#778”](#)中跟踪。

使用 WPF 进行数据绑定

此分步演练演示如何以“主详细信息”形式将 POCO 类型绑定到 WPF 控件。应用程序使用实体框架 API 使用数据库中的数据填充对象、跟踪更改并将数据保存到数据库中。

模型定义了参与一对多关系的两种类型：类别（主体\主控形状）和产品（相关\详细信息）。然后，Visual Studio 工具用于将模型中定义的类型绑定到 WPF 控件。WPF 数据绑定框架支持相关对象之间的导航：选择主视图中的行会导致详细视图使用相应的子数据进行更新。

本演练中的屏幕截图和代码列表取自 Visual Studio 2013，但您可以使用 Visual Studio 2012 或 Visual Studio 2010 完成本演练。

使用“对象”选项创建 WPF 数据源

对于以前版本的实体框架，我们曾建议在基于使用 EF 设计器创建的模型创建新数据源时使用数据库选项。这是因为设计器将生成从 `ObjectContext` 派生的上下文和派生自实体对象的实体类。使用“数据库”选项将帮助您编写用于与此 API 曲面交互的最佳代码。

2012 年 Visual Studio 的 EF 设计人员和 2013 年可视化工作室生成源自 `DbContext` 的上下文以及简单的 POCO 实体类。使用 Visual Studio 2010，我们建议交换到代码生成模板，该模板使用 `DbContext`，如本演练后面的所述。

使用 `DbContext` API 曲面时，应在创建新数据源时使用 Object 选项，如本演练所示。

如果需要，您可以为使用 EF 设计器创建的模型[还原到基于 ObjectContext 的代码生成](#)。

先决条件

您需要安装 Visual Studio 2013、Visual Studio 2012 或 Visual Studio 2010 才能完成本演练。

如果您使用的是 Visual Studio 2010，则还必须安装 NuGet。有关详细信息，请参阅[安装 NuGet](#)。

创建应用程序

- 打开 Visual Studio
- 文件>->新建 - 项目...
- 在左侧窗格中选择 Windows，在右侧窗格中选择 WPF 应用程序
- 输入 WPF 与 EFSample 作为名称
- 选择“确定”

安装实体框架 NuGet 包

- 在解决方案资源管理器中，右键单击 WinForms 与 EFSample 项目

- 选择 "管理 NuGet 包..." ...
- 在"管理 NuGet 包"对话框中, 选择 "联机"选项卡并选择 "实体框架"包
- 单击"安装"

NOTE

除了实体框架程序集外, 还添加了对 System.组件模型.Data注释的引用。如果项目具有对 System.Data.Entity 的引用, 则在安装实体框架包时将删除该项目。System.Data.实体程序集不再用于实体框架 6 应用程序。

定义模型

在本演练中, 您可以选择使用代码优先或 EF 设计器实现模型。完成以下两节之一。

选项 1: 首先使用代码定义模型

本节演示如何使用 Code First 创建模型及其关联的数据库。如果您希望使用数据库优先使用 EF 设计器从数据库反向工程模型, 请跳到下一部分(选项 2: 使用数据库优先定义模型)

使用代码优先开发时, 通常先编写定义概念(域)模型的 .NET 框架类。

- 向 WPFwithEFSample 添加新类:
 - 右键单击项目名称
 - 选择 "添加", 然后选择"新建项目"
 - 选择 "类" 并输入类名称 **的产品**
- 将 **产品** 类定义替换为以下代码:

```
namespace WPFwithEFSample
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }

        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}

- Add a **Category** class with the following definition:

using System.Collections.ObjectModel;

namespace WPFwithEFSample
{
    public class Category
    {
        public Category()
        {
            this.Products = new ObservableCollection<Product>();
        }

        public int CategoryId { get; set; }
        public string Name { get; set; }

        public virtual ObservableCollection<Product> Products { get; private set; }
    }
}
```

产品类上的 "产品" 类和 "类别" 属性是导航属性。Products 在实体框架中, 导航属性提供了一种导航两种实体类型之间的关系的方法。

除了定义实体之外，还需要定义派生自 DbContext 并公开 DbSet< TEntity > 属性的类。 DbSet< TEntity > 属性让上下文知道要在模型中包括哪些类型。

DbContext 派生类型的实例在运行时管理实体对象，其中包括将对象与数据库中的数据填充、更改跟踪和将数据保存到数据库。

- 向具有以下定义的项目添加新的 ProductContext 类：

```
using System.Data.Entity;

namespace WPFwithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}
```

编译该项目。

选项 2：首先使用数据库定义模型

本节演示如何使用数据库 First 使用 EF 设计器从数据库反向工程模型。如果完成了上一节（选项 1：使用代码优先定义模型），则跳过此部分并直接转到“延迟加载”部分。

创建现有数据库

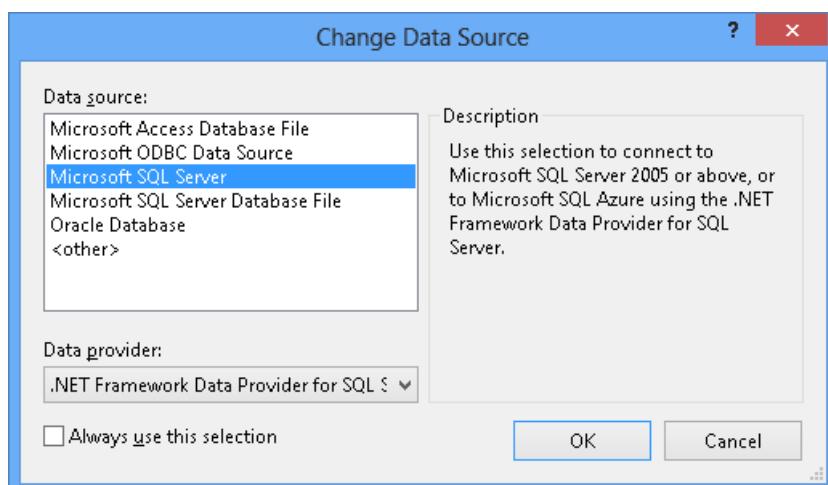
通常，当您将现有数据库作为目标时，该数据库将创建，但本演练需要创建要访问的数据库。

与 Visual Studio 一起安装的数据库服务器因已安装的 Visual Studio 版本而异：

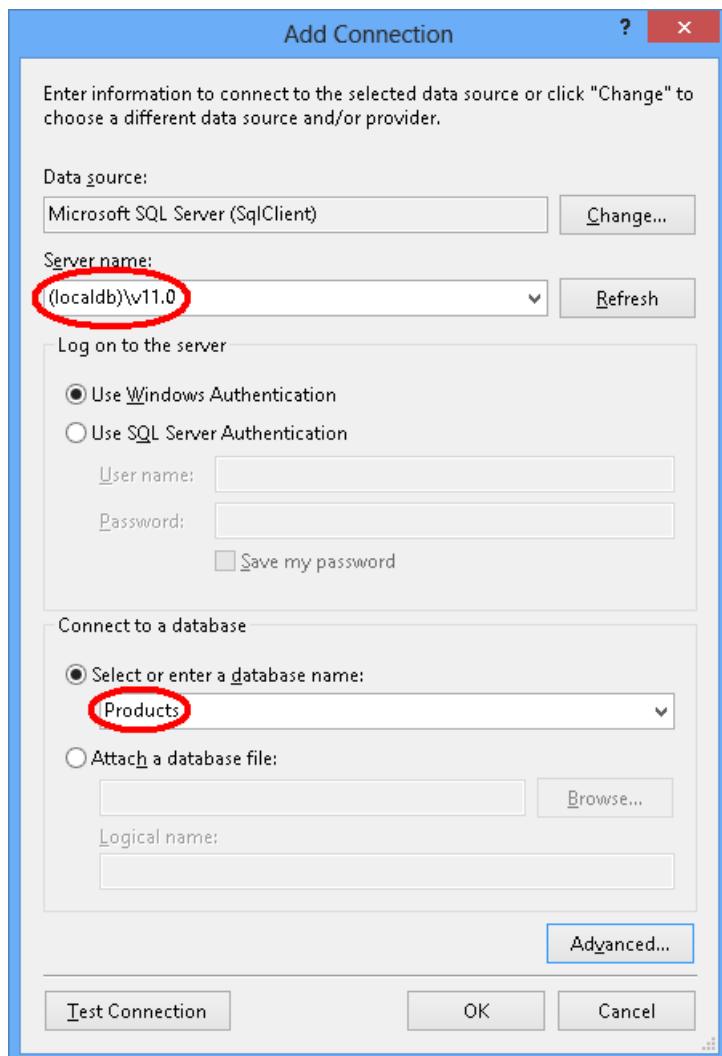
- 如果您使用的是 Visual Studio 2010，您将创建一个 SQL Express 数据库。
- 如果您使用的是 Visual Studio 2012，则您将创建一个本地数据库数据库。

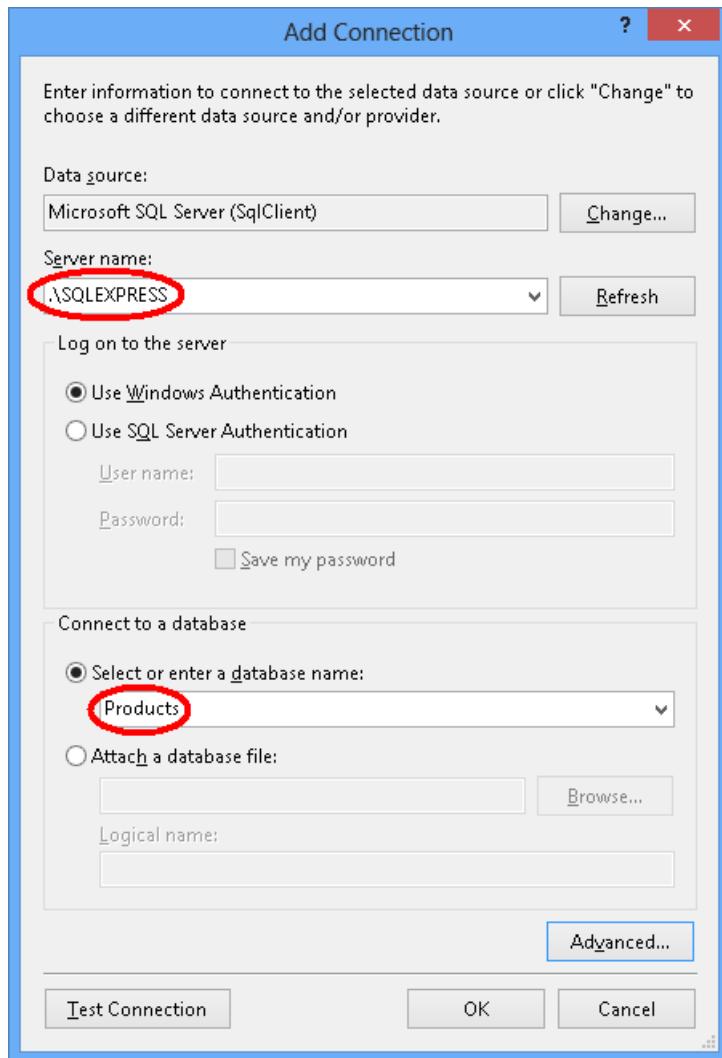
让我们继续生成数据库。

- 视图 -> 服务器资源管理器
- 右键单击数据连接 -> 添加连接...
- 如果尚未从服务器资源管理器连接到数据库，则需要选择 Microsoft SQL Server 作为数据源



- 连接到本地数据库或 SQL Express，具体取决于已安装的哪个，然后输入“产品”作为数据库名称





- 选择 "确定", 系统将询问您是否要创建新数据库, 请选择"是"



- 新数据库现在将显示在服务器资源管理器中, 右键单击它并选择 "新查询"
- 将以下 SQL 复制到新查询中, 然后右键单击查询并选择 "执行"

```

CREATE TABLE [dbo].[Categories] (
    [CategoryId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    CONSTRAINT [PK_dbo.Categories] PRIMARY KEY ([CategoryId])
)

CREATE TABLE [dbo].[Products] (
    [ProductId] [int] NOT NULL IDENTITY,
    [Name] [nvarchar](max),
    [CategoryId] [int] NOT NULL,
    CONSTRAINT [PK_dbo.Products] PRIMARY KEY ([ProductId])
)

CREATE INDEX [IX_CategoryId] ON [dbo].[Products]([CategoryId])

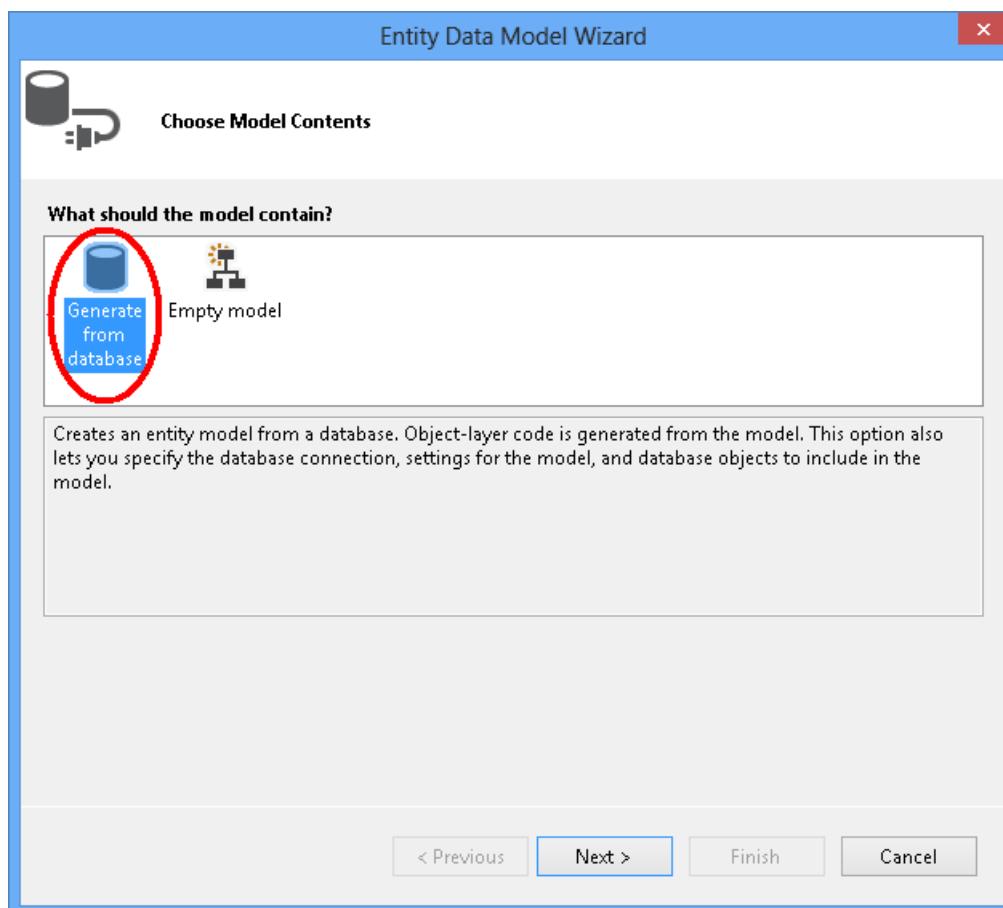
ALTER TABLE [dbo].[Products] ADD CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY
([CategoryId]) REFERENCES [dbo].[Categories] ([CategoryId]) ON DELETE CASCADE

```

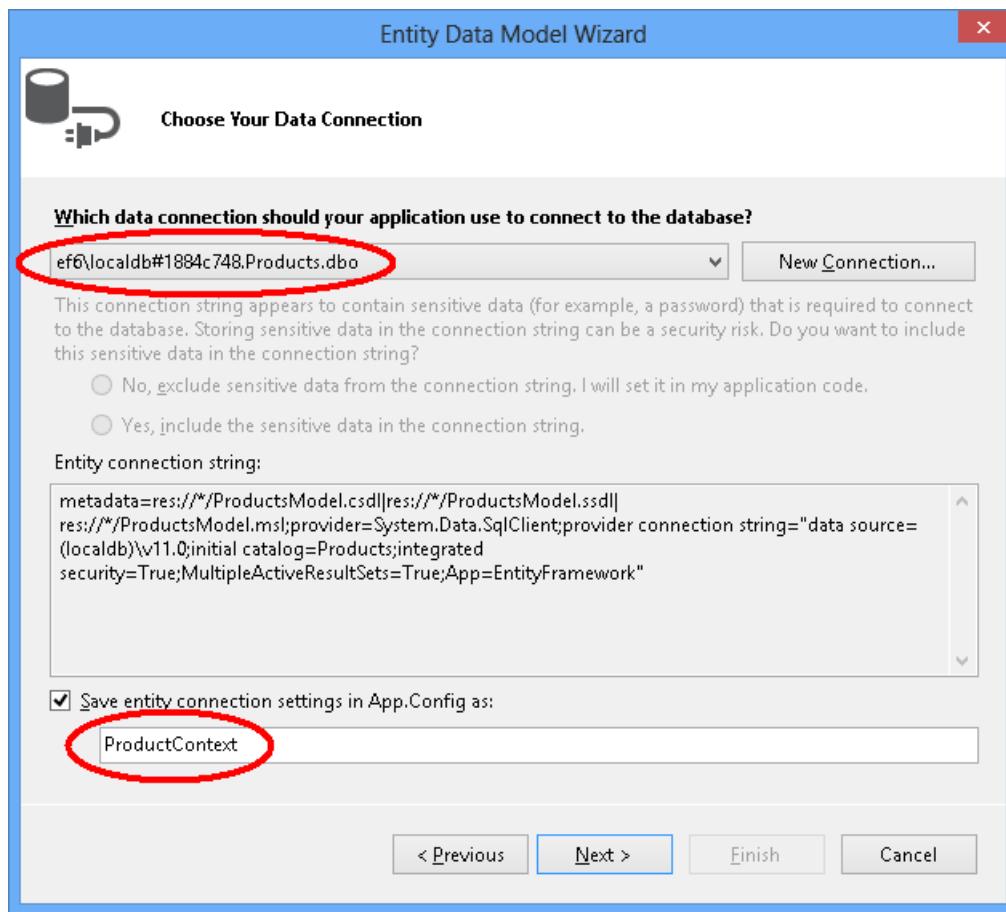
反向工程模型

我们将利用实体框架设计器(作为Visual Studio的一部分)创建我们的模型。

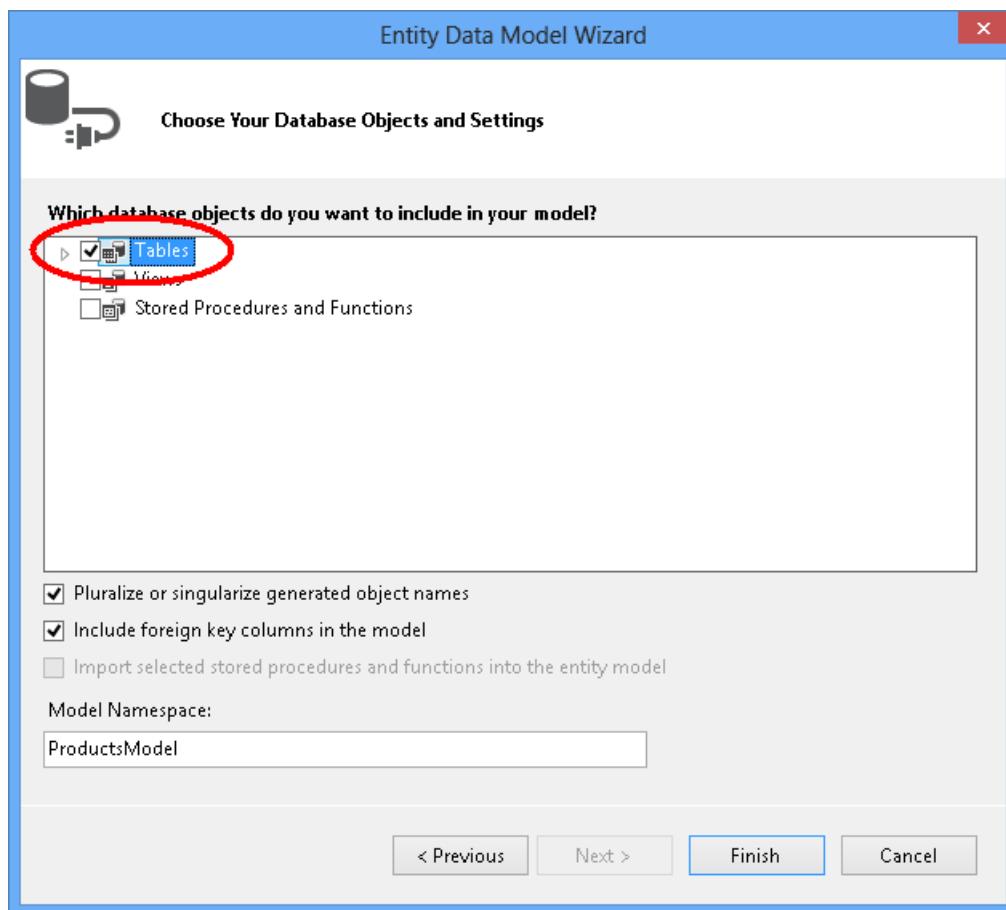
- 项目>-添加新项目...
- 从左侧菜单中选择**数据**,然后**ADO.NET实体数据模型**
- 输入**产品模型**作为名称,然后单击**"确定"**
- 这将启动**实体数据模型向导**
- 选择**"从数据库生成"**,然后单击**"下一步"**



- 选择与在第一部分中创建的数据库的连接,输入**ProductContext**作为连接字符串的名称,然后单击**"下一步"**



- 单击"表格"旁边的复选框导入所有表, 然后单击"完成"



反向工程过程完成后, 新模型将添加到项目中, 并打开, 供您在实体框架设计器中查看。App.config 文件也已添加到项目中, 并包含数据库的连接详细信息。

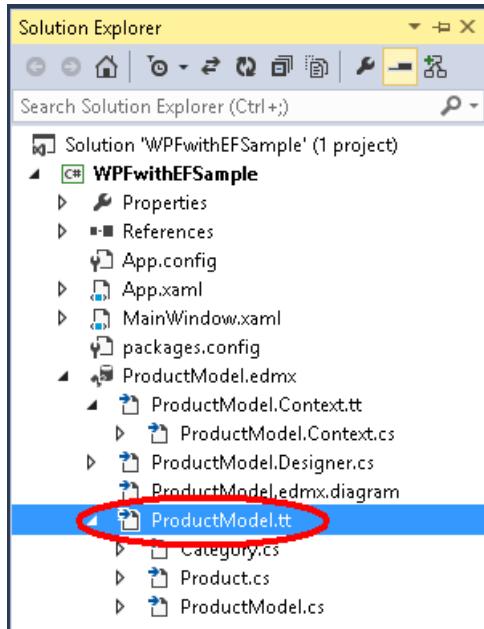
如果您在 Visual Studio 2010 中工作，则需要更新 EF 设计器才能使用 EF6 代码生成。

- 右键单击 EF 设计器中模型的空白点，然后选择“添加代码生成项...”
- 从左侧菜单中选择“联机模板”并搜索 DbContext
- 选择 EF 6.x DbContext#生成器为 C，输入产品模型作为名称，然后单击“添加”

更新数据绑定的代码生成

EF 使用 T4 模板从模型生成代码。视觉工作室附带或从 Visual Studio 库下载的模板用于一般用途。这意味着从这些模板生成的实体具有简单的 ICollection<T> 属性。但是，使用 WPF 执行数据绑定时，最好将可观察集合用于集合属性，以便 WPF 可以跟踪对集合所做的更改。为此，我们将修改模板以使用可观察集合。

- 打开解决方案资源管理器并查找产品模型.edmx 文件
- 查找将嵌套在 ProductModel.edmx 文件下的 ProductModel.tt 文件



- 双击 ProductModel.tt 文件以在可视化工作室编辑器中打开该文件
- 查找 "ICollection" 的两个事件并将其替换为“可观察集合”。这些位于大约 296 和 484 行。
- 查找“哈希集”的第一次出现与“可观察集合”替换。此事件大约位于第 50 行。不要替换代码后面找到的第二个哈希集。
- 查找“系统.集合.通用”的唯一匹配项并将其替换为“系统.集合.objectModel”。这大约位于第 424 行。
- 保存 ProductModel.tt 文件。这应会导致重新生成实体的代码。如果代码未自动重新生成，则右键单击 ProductModel.tt 并选择“运行自定义工具”。

如果您现在打开 Category.cs 文件（嵌套在 ProductModel.tt 下），那么您应该看到产品集合具有“可观察<集合产品>”的类型。

编译该项目。

延迟加载

产品类上的“产品”类和“类别”属性是导航属性。Products 在实体框架中，导航属性提供了一种导航两种实体类型之间的关系的方法。

EF 为您提供了在首次访问导航属性时自动从数据库加载相关实体的选项。使用这种类型的加载（称为延迟加载），请注意，如果内容尚未在上下文中，则第一次访问每个导航属性时，将对数据库执行单独的查询。

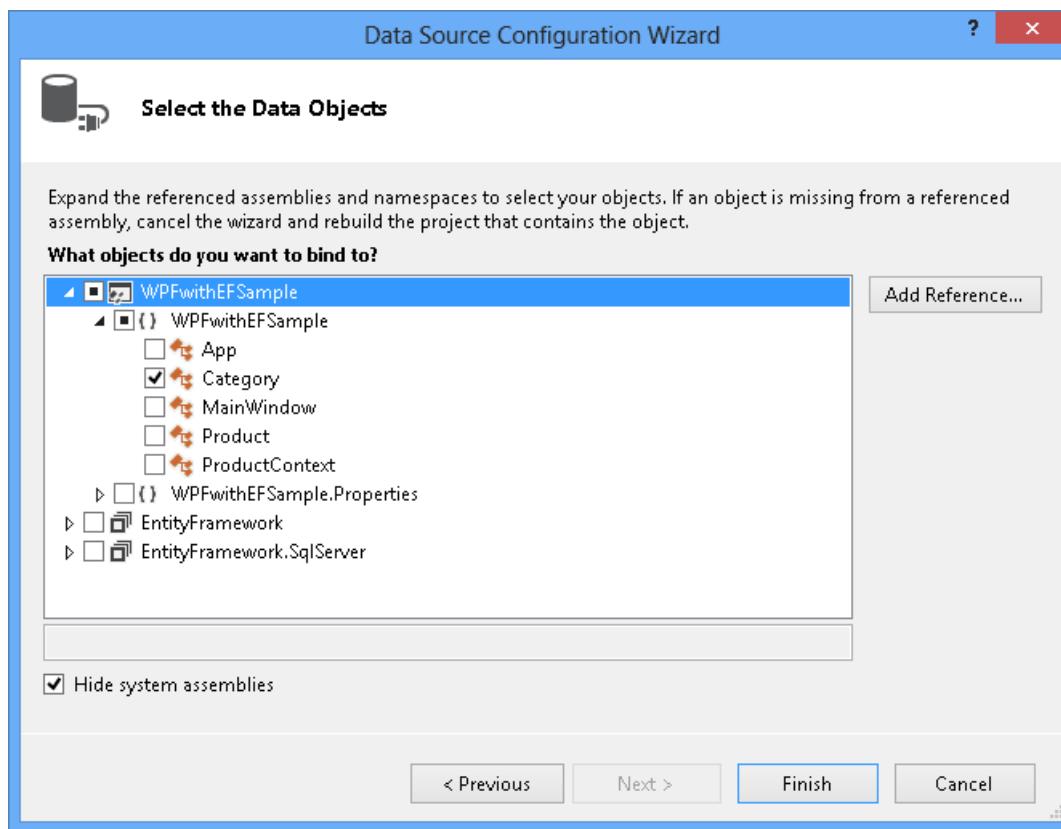
使用 POCO 实体类型时，EF 通过在运行时创建派生代理类型的实例，然后在类中重写虚拟属性来添加加载挂钩，从

而实现延迟加载。要获取相关对象的延迟加载，必须声明导航属性 getter 为公共和虚拟（在 Visual Basic 中可重写），并且不得密封类（在 Visual Basic 中不可重写）。使用数据库时，第一个导航属性会自动成为虚拟属性，以启用延迟加载。在“代码第一”部分中，我们选择出于同样的原因使导航属性成为虚拟

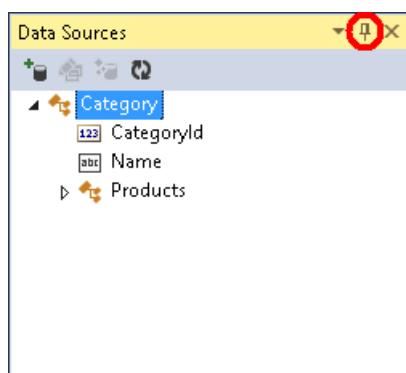
将对象绑定到控件

将模型中定义的类添加为此 WPF 应用程序的数据源。

- 双击解决方案资源管理器中的MainWindow.xaml以打开主窗体
- 从主菜单中，选择“项目 > - 添加新数据源...”（在 Visual Studio 2010 中，您需要选择数据 -> 添加新数据源...）
- 在“选择数据源类型窗口”中，选择对象并单击“下一步”
- 在“选择数据对象”对话框中，展开WPF与EFSample两次并选择“类别”
无需选择产品数据源，因为我们将通过“产品****”在类别数据源上的属性获取它



- 单击“完成”。
- 如果“数据源”窗口未显示，则选择“查看 -> 其他> Windows- 数据源”，在 MainWindow.xaml 窗口旁边打开数据源。
- 按引脚图标，以便数据源窗口不会自动隐藏。如果窗口已可见，则可能需要点击刷新按钮。



- 选择 "类别" 数据源并将其拖动到窗体上。

当我们拖动此源时发生了以下情况：

- 类别 ViewSource 资源和类别数据网格控件已添加到 XAML
- 父网格元素上的 DataContext 属性设置为 "[静态资源类别 ViewSource]"。类别 ViewSource 资源用作外部\父网格元素的绑定源。然后，内部网格元素从父网格继承 DataContext 值(类别 DataGrid 的 ItemsSource 属性设置为 "{绑定}")

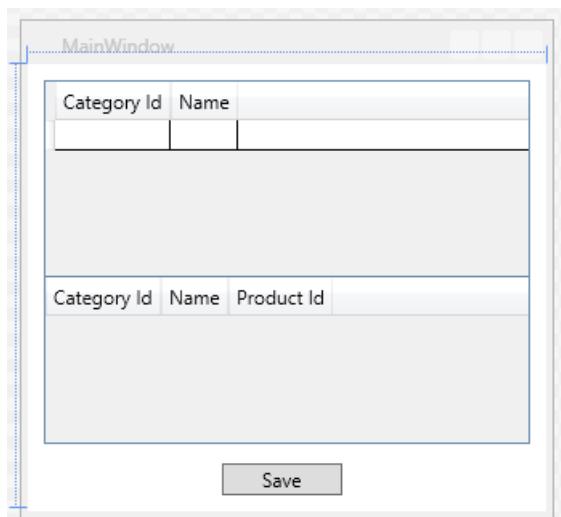
```
<Window.Resources>
    <CollectionViewSource x:Key="categoryViewSource"
        d:DesignSource="{d:DesignInstance {x:Type local:Category}, CreateList=True}"/>
</Window.Resources>
<Grid DataContext="{StaticResource categoryViewSource}">
    <DataGrid x:Name="categoryDataGrid" AutoGenerateColumns="False" EnableRowVirtualization="True"
        ItemsSource="{Binding}" Margin="13,13,43,191"
        RowDetailsVisibilityMode="VisibleWhenSelected">
        <DataGrid.Columns>
            <DataGridTextColumn x:Name="categoryIdColumn" Binding="{Binding CategoryId}"
                Header="Category Id" Width="SizeToHeader"/>
            <DataGridTextColumn x:Name="nameColumn" Binding="{Binding Name}"
                Header="Name" Width="SizeToHeader"/>
        </DataGrid.Columns>
    </DataGrid>
</Grid>
```

添加详细信息网格

现在，我们有一个网格来显示类别，让我们添加一个详细信息网格来显示关联的产品。

- 从 "类别" 数据源下选择 "产品" 属性，并将其拖动到窗体上。
 - 类别 ProductsViewSource 资源和产品数据网格将添加到 XAML
 - 此资源的绑定路径设置为产品
 - WPF 数据绑定框架可确保产品 DataGrid 中仅显示与所选类别相关的产品
- 从"工具箱"中，将按钮拖动到窗体。将 "名称" 属性设置为按钮"保存"，将 "内容" 属性设置为 "保存"。

窗体应如下所示：

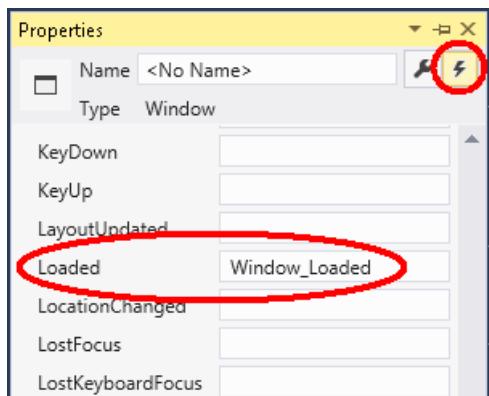


添加处理数据交互的代码

是时候将一些事件处理程序添加到主窗口了。

- 在 XAML 窗口中，单击**<窗口**元素，这将选择主窗口

- 在“属性”窗口中选择右上角的事件，然后双击“已加载”标签右侧的文本框



- 此外，通过双击设计器中的“保存”按钮，添加“保存”按钮的 Click 事件。

这带来了表单背后的代码，我们现在将编辑代码以使用 `ProductContext` 执行数据访问。更新主窗口的代码，如下所示。

该代码声明产品上下文的长期运行实例。`ProductContext` 对象用于查询数据并将其保存到数据库中。然后，从重写的 `OnClose` 方法调用 `ProductContext` 实例上的 `Dispose()`。代码注释提供有关代码功能的详细信息。

```

using System.Data.Entity;
using System.Linq;
using System.Windows;

namespace WPFwithEFSample
{
    public partial class MainWindow : Window
    {
        private ProductContext _context = new ProductContext();
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            System.Windows.Data.CollectionViewSource categoryViewSource =
                ((System.Windows.Data.CollectionViewSource)(this.FindResource("categoryViewSource")));

            // Load is an extension method on IQueryable,
            // defined in the System.Data.Entity namespace.
            // This method enumerates the results of the query,
            // similar to ToList but without creating a list.
            // When used with Linq to Entities this method
            // creates entity objects and adds them to the context.
            _context.Categories.Load();

            // After the data is loaded call the DbSet<T>.Local property
            // to use the DbSet<T> as a binding source.
            categoryViewSource.Source = _context.Categories.Local;
        }

        private void buttonSave_Click(object sender, RoutedEventArgs e)
        {
            // When you delete an object from the related entities collection
            // (in this case Products), the Entity Framework doesn't mark
            // these child entities as deleted.
            // Instead, it removes the relationship between the parent and the child
            // by setting the parent reference to null.
            // So we manually have to delete the products
            // that have a Category reference set to null.
        }
    }
}

```

```

// The following code uses LINQ to Objects
// against the Local collection of Products.
// The ToList call is required because otherwise the collection will be modified
// by the Remove call while it is being enumerated.
// In most other situations you can use LINQ to Objects directly
// against the Local property without using ToList first.
foreach (var product in _context.Products.Local.ToList())
{
    if (product.Category == null)
    {
        _context.Products.Remove(product);
    }
}

_context.SaveChanges();
// Refresh the grids so the database generated values show up.
this.categoryDataGrid.Items.Refresh();
this.productsDataGrid.Items.Refresh();
}

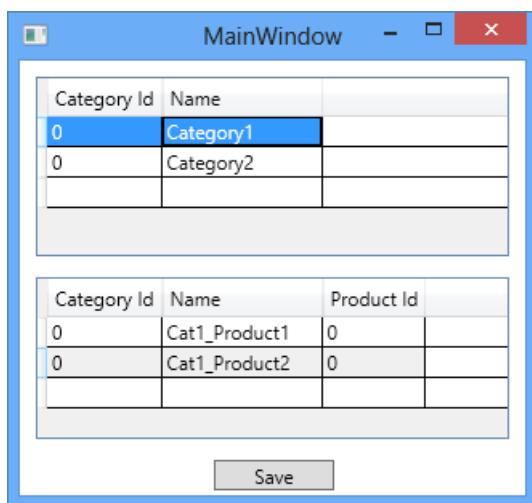
protected override void OnClosing(System.ComponentModel.CancelEventArgs e)
{
    base.OnClosing(e);
    this._context.Dispose();
}
}

}

```

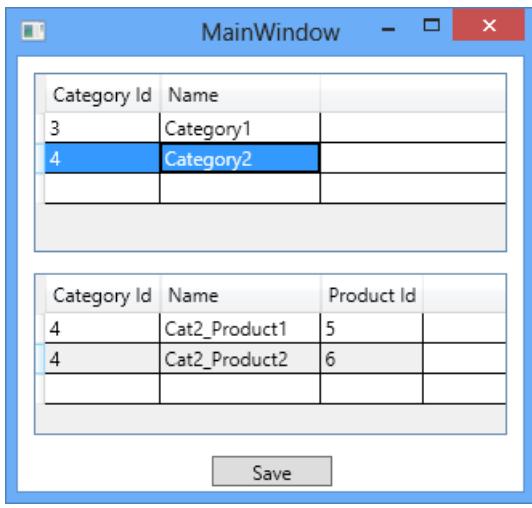
测试 WPF 应用程序

- 编译并运行该应用程序。如果您先使用代码，您将看到为您创建了WPFwithEFSample.ProductContext数据库。
- 在顶部网格中输入类别名称，在底部网格中输入产品名称不要在 ID 列中输入任何内容，因为主键由数据库生成



- 按“保存”按钮将数据保存到数据库

调用 DbContext 的 `SaveChanges()` 后，使用数据库生成的值填充 ID。因为我们在 `SaveChanges()` 后调用 `Refresh()`，`DataGrid` 控件也使用新值进行更新。



其他资源

要了解有关使用 WPF 绑定到集合的数据的详细信息，请参阅 WPF 文档中的[本主题](#)。

使用断开连接的实体

2020/4/8 ·

在基于实体框架的应用程序中，上下文类负责检测应用于已跟踪实体的更改。调用 `SaveChanges` 方法会将上下文跟踪的更改保存到数据库。使用 n 层应用程序时，实体对象通常会在从上下文断开连接时发生变动，且必须决定如何跟踪更改并向上下文报告这些更改。本主题讨论使用实体框架和断开连接的实体时不同的可用选项。

Web 服务框架

Web 服务技术通常支持可用于在各个断开连接的对象上保存更改的模式。例如，通过 ASP.NET Web API 可编写控制器操作，这些操作可以包括对 EF 的调用，以保存对数据库上的对象所做的更改。实际上，Visual Studio 中的 Web API 工具可以轻松地从实体框架 6 模型构建 Web API 控制器。有关详细信息，请参阅[一起使用 Web API 和实体框架 6](#)。

以前，还有其他一些 Web 服务技术提供与实体框架的集成，例如 [WCF 数据服务](#) 和 [RIA 服务](#)。

低级别 EF API

如果不想使用现有的 n 层解决方案，或者想要自定义 Web API 服务中控制器操作内发生的内容，则可通过实体框架提供的 API 应用在断开连接的层上所做的更改。有关详细信息，请参阅[添加、附加和实体状态](#)。

自跟踪实体

在从 EF 上下文断开连接时，跟踪实体的任意图上的更改是一个难题。可采用自跟踪实体代码生成模板来尝试解决。此模板生成实体类作为实体本身的状态，该类包含可跟踪断开连接的层上所做更改的逻辑。还会生成一组扩展方法，将这些更改应用于上下文。

此模板可与使用 EF 设计器创建的模型一起使用，但不能与 Code First 模型一起使用。有关详细信息，请参阅[自跟踪实体](#)。

IMPORTANT

我们不再建议使用自跟踪实体模板。它将仅继续用于支持现有应用程序。如果应用程序需要使用断开连接的实体图，请考虑其他替代方案，例如[可跟踪实体](#)，它与自跟踪实体类似，社区在更积极地开发这种技术，或使用低级别更改跟踪 API 编写自定义代码。

自跟踪实体

2020/4/8 •

IMPORTANT

我们不再建议使用自跟踪实体模板。它将仅继续用于支持现有应用程序。如果应用程序需要使用断开连接的实体图，请考虑其他替代方案，例如[可跟踪实体](#)，它与自跟踪实体类似，社区在更积极地开发这种技术，或使用低级别更改跟踪 API 编写自定义代码。

在基于实体框架的应用程序中，上下文负责跟踪对象中的更改。然后，使用 `SaveChanges` 方法将更改保存到数据库。使用 N 层应用程序时，实体对象通常会从上下文断开连接，用户必须决定如何跟踪更改并向上下文报告这些更改。自跟踪实体 (STE) 有助于跟踪任意层中的更改，之后可将这些更改重放入上下文中进行保存。

仅当上下文在更改对象图所在的层上不可用时才使用 STE。如果上下文可用，则无需使用 STE，因为上下文将负责跟踪更改。

此模板项会生成两个 .tt(文本模板)文件：

- `model name<.tt` 文件可生成实体类型、包含自跟踪实体使用的更改跟踪逻辑的 `Helper` 类以及允许设置自跟踪实体的状态的扩展方法>。
- `model name<.Context.tt` 文件生成派生的上下文和扩展类，该类包含 `ObjectContext` 和 `ObjectSet` 类的 `ApplyChanges` 方法>。这些方法检查自跟踪实体图中包含的更改跟踪信息，以推断在数据库中保存这些更改所必须执行的一组操作。

入门

若要开始操作，请访问[自跟踪实体演练](#)页。

使用自跟踪实体时的功能性注意事项

IMPORTANT

我们不再建议使用自跟踪实体模板。它将仅继续用于支持现有应用程序。如果应用程序需要使用断开连接的实体图，请考虑其他替代方案，例如[可跟踪实体](#)，它与自跟踪实体类似，社区在更积极地开发这种技术，或使用低级别更改跟踪 API 编写自定义代码。

使用自跟踪实体时应考虑以下事项：

- 确保您的客户端项目具有对包含实体类型的程序集的引用。如果您只将服务引用添加到客户端项目，则客户端项目将使用 WCF 代理类型，而不是实际的自跟踪实体类型。这意味着您将不能获得管理客户端上实体跟踪的自动通知功能。如果您不希望包含实体类型，则必须手动设置客户端的更改跟踪信息以便将更改发送回服务。
- 对服务操作的调用应是无状态的，并且创建一个新的对象上下文实例。同样建议在 `using` 块中创建对象上下文。
- 如果在将客户端上已修改的图发送到服务后想要在客户端上继续使用同一张图，则必须手动循环访问该图，并对每个对象调用 `AcceptChanges` 方法以重置更改跟踪器。

如果图中的对象包含具有数据库生成的值（例如，标识值或并发值）的属性，则在调用 `SaveChanges` 方法后，实体框架将把这些属性的值替换为数据库生成的值。可实现服务操作以返回保存的对象或生成的

属性值列表，从而将对象发送回客户端。然后，客户端需要将对象实例或对象属性值替换为从服务操作返回的对象或属性值。

- 合并多个服务请求的关系图可能会在生成的关系图中引入具有重复键值的对象。调用 `ApplyChanges` 方法时，实体框架不会删除具有重复键的对象，但会引发异常。为了避免生成具有重复键值的图，请遵循以下博客中所述的模式之一：[自跟踪实体: `ApplyChanges` 和重复实体](#)。
- 当您通过设置外键属性更改对象之间的关系时，引用导航属性设置为 `Null`，并且不同步到客户端上的相应主体实体。将图附加到对象上下文之后（例如，调用 `ApplyChanges` 方法之后），会同步外键属性和导航属性。

如果您已对外键关系指定级联删除，则引用导航属性与相应主体对象不同步会是个问题。如果您删除了主体，则这种删除将不会传播到依赖对象。如果您已指定级联删除，则使用导航属性更改关系，而不是设置外键属性。

- 执行延迟加载时不启用自跟踪实体。
- 自跟踪实体不支持二进制序列化和针对 ASP.NET 状态管理对象的序列化。但是，您可以自定义此模板来添加二进制序列化支持。有关详细信息，请参阅[将二进制序列化和 ViewState 用于自跟踪实体](#)。

安全注意事项

使用自跟踪实体时，应考虑以下安全注意事项：

- 服务不应信任从不受信任的客户端或通过不受信任的通道检索或更新数据的请求。客户端必须经身份验证：应使用安全通道或消息信封。必须验证客户端更新或检索数据的请求，从而确保这些请求符合对于给定方案合法的预期更改。
- 避免将敏感信息用作实体键（例如身份证号码）。这样可降低意外地将自跟踪实体图中的敏感信息序列化到不完全可信的客户端的可能性。通过使用独立关联，实体中与正在序列化的键相关的原始键也可能会发送到客户端。
- 为了避免向客户端层传播包含敏感数据的异常消息，应将服务器层上对 `ApplyChanges` 和 `SaveChanges` 的调用包装到异常处理代码中。

自跟踪实体演练

2020/3/11 •

IMPORTANT

我们不再建议使用自跟踪实体模板。它将仅继续用于支持现有应用程序。如果应用程序需要使用断开连接的实体图，请考虑其他替代方案，例如[可跟踪实体](#)，它与自跟踪实体类似，社区在更积极地开发这种技术，或使用低级别更改跟踪 API 编写自定义代码。

本演练演示了 Windows Communication Foundation (WCF) 服务公开返回实体关系图的操作的情况。接下来，客户端应用程序操作该图并将修改提交给使用实体框架验证和保存数据库更新的服务操作。

在完成本演练之前，请务必阅读 "[自跟踪实体](#)" 页。

此演练完成以下操作：

- 创建要访问的数据库。
- 创建包含模型的类库。
- 交换到 "自跟踪实体生成器" 模板。
- 将实体类移到单独的项目中。
- 创建一个 WCF 服务，该服务公开用于查询和保存实体的操作。
- 创建使用服务的客户端应用程序(控制台和 WPF)。

我们将在本演练中使用 Database First，但相同的技术同样适用于 Model First。

先决条件

若要完成本演练，你将需要最新版本的 Visual Studio。

创建数据库

随 Visual Studio 一起安装的数据库服务器因安装的 Visual Studio 版本而异：

- 如果使用的是 Visual Studio 2012，则将创建一个 LocalDB 数据库。
- 如果使用的是 Visual Studio 2010，则将创建 SQL Express 数据库。

接下来，生成数据库。

- 打开 Visual Studio
- 视图-> 服务器资源管理器
- 右键单击 "数据连接-> 添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择Microsoft SQL Server作为数据源
- 连接到 LocalDB 或 SQL Express，具体取决于你安装的是哪个
- 输入STESample作为数据库名称
- 选择 "确定"，系统会询问您是否要创建新数据库，请选择 "是"
- 新数据库现在将出现在服务器资源管理器
- 如果使用的是 Visual Studio 2012
 - 在服务器资源管理器中右键单击该数据库，然后选择 "新建查询"
 - 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择 "执行"

- 如果使用的是 Visual Studio 2010
 - 选择数据>Transact-sql 编辑器->新建查询连接 ...
 - 输入。\\SQLEXPRESS作为服务器名称, 然后单击"确定"
 - 从"查询编辑器"顶部的下拉菜单中选择"STESample"数据库
 - 将以下 SQL 复制到新的查询中, 然后右键单击该查询, 然后选择"执行 SQL"。

```

CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId])
    ON DELETE CASCADE
);

SET IDENTITY_INSERT [dbo].[Blogs] ON
INSERT INTO [dbo].[Blogs] ([BlogId], [Name], [Url]) VALUES (1, N'ADO.NET Blog', N'blogs.msdn.com/adonet')
SET IDENTITY_INSERT [dbo].[Blogs] OFF
INSERT INTO [dbo].[Posts] ([Title], [Content], [BlogId]) VALUES (N'Intro to EF', N'Interesting stuff...', 1)
INSERT INTO [dbo].[Posts] ([Title], [Content], [BlogId]) VALUES (N'What is New', N'More interesting stuff...', 1)
  
```

创建模型

首先, 我们需要一个项目来放置模型。

- 文件->>项目..。
- 从左窗格中选择"Visual C#", 然后选择"类库"
- 输入STESample作为名称, 然后单击"确定"

现在, 我们将在 EF 设计器中创建一个简单的模型来访问数据库:

- 项目->"添加新项..."
- 从左窗格中选择"数据", 然后ADO.NET 实体数据模型
- 输入BloggingModel作为名称, 然后单击"确定"
- 选择"从数据库生成", 然后单击"下一步"
- 输入在上一部分中创建的数据库的连接信息
- 输入"bloggingcontext"作为连接字符串的名称, 然后单击"下一步"
- 选中"表"旁边的框, 然后单击"完成"

交换到粘贴代码生成

现在, 我们需要禁用默认代码生成并交换到自跟踪实体。

如果使用的是 Visual Studio 2012

- 展开解决方案资源管理器中的BloggingModel, 删除BloggingModel.tt和BloggingModel.Context.tt 这将禁用默认代码生成
- 右键单击 EF 设计器图面上的空白区域, 然后选择"添加代码生成项..."

- 从左窗格中选择 "联机", 然后搜索粘贴生成器
- 选择粘贴生成器 For C# 模板, 输入STETemplate作为名称, 然后单击 "添加"
- STETemplate.tt和STETemplate.Context.tt文件添加到了 BloggingModel 文件下

如果使用的是 Visual Studio 2010

- 右键单击 EF 设计器图面上的空白区域, 然后选择 "添加代码生成项 ... "
- 从左窗格中选择 "代码", 然后ADO.NET 自跟踪实体生成器
- 输入STETemplate作为名称, 然后单击 "添加"
- STETemplate.tt和STETemplate.Context.tt文件会直接添加到你的项目

将实体类型移到单独的项目中

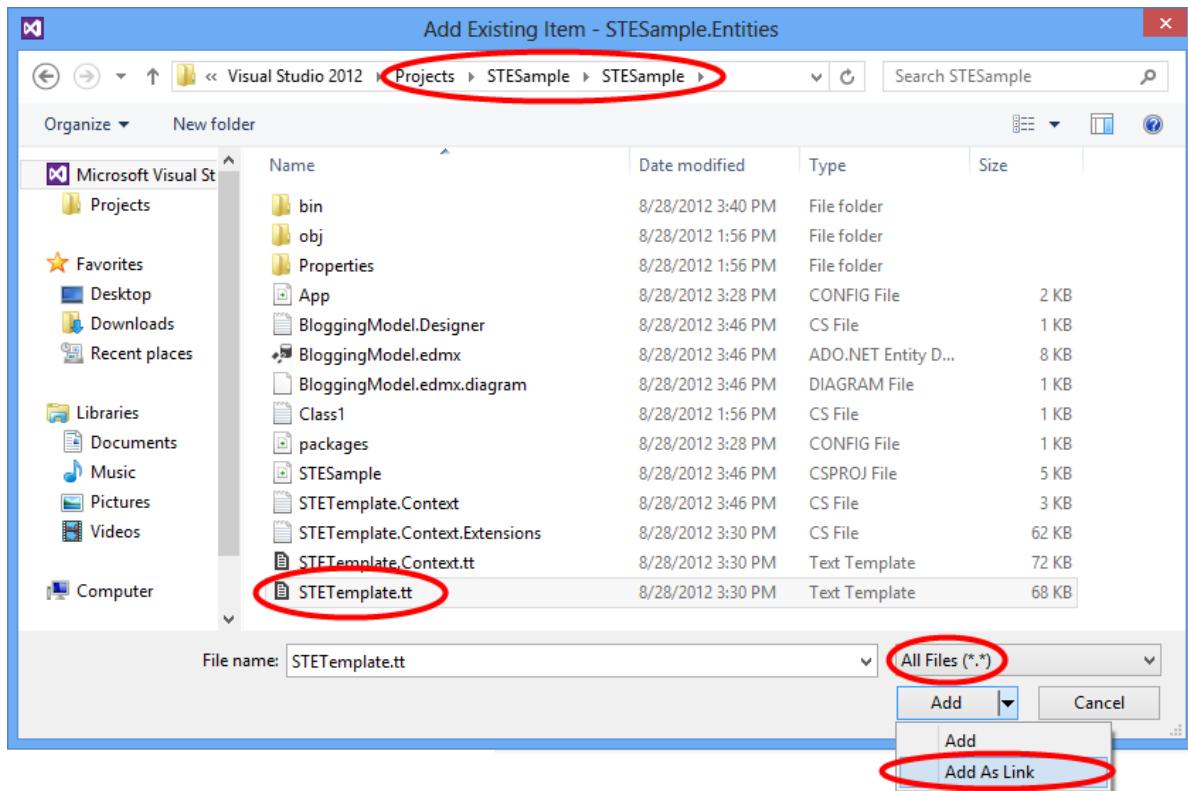
若要使用自跟踪实体, 客户端应用程序需要访问模型中生成的实体类。由于我们不想将整个模型公开给客户端应用程序, 因此我们要将实体类移到单独的项目中。

第一步是停止生成现有项目中的实体类:

- 在解决方案资源管理器中右键单击 "STETemplate.tt ", 然后选择 "属性"
- 在 "属性" 窗口中, 清除 "CustomTool" 属性中的TextTemplatingFileGenerator
- 在解决方案资源管理器中展开 "STETemplate.tt ", 并删除其下嵌套的所有文件

接下来, 我们将添加一个新项目, 并在其中生成实体类

- 文件-> 添加> 项目 ..。
- 从左窗格中选择 "Visual C#", 然后选择" 类库 "
- 输入STESample作为名称, 然后单击 "确定"
- 项目-> 添加现有项 ..。
- 导航到STESample项目文件夹
- 选择查看所有文件(**)
- 选择STETemplate.tt文件
- 单击 "添加" 按钮旁边的下拉箭头, 然后选择 "添加为链接"。



我们还将确保在上下文相同的命名空间中生成实体类。这只是减少了我们需要在应用程序中添加的 using 语句的数量。

- 在解决方案资源管理器中右键单击链接的STETemplate.tt，然后选择“属性”
- 在“属性”窗口中，将自定义工具命名空间设置为STESample

粘贴模板生成的代码将需要引用 system.exception 才能进行编译。在可序列化实体类型上使用的 WCF DataContract 和DataMember 特性需要此库。

- 右键单击解决方案资源管理器中的STESample项目，然后选择“添加引用...”
 - 在 Visual Studio 2012 中，选中“system.web”旁边的框，然后单击“确定”
 - 在 Visual Studio 2010 中，选择“System.web”并单击“确定”

最后，具有中的上下文的项目将需要对实体类型的引用。

- 在解决方案资源管理器中右键单击“STESample”项目，然后选择“添加引用...”
 - 在 Visual Studio 2012 中，从左窗格中选择“解决方案”，选中“STESample”旁边的框，然后单击“确定”
 - 在 Visual Studio 2010 中，选择“项目”选项卡，选择“STESample”，然后单击“确定”

NOTE

将实体类型移动到单独的项目的另一种方法是移动模板文件，而不是将其链接到其默认位置。如果执行此操作，则需要更新模板中的 inputFile 变量，以提供 edmx 文件的相对路径（在本示例中，为 ..\BloggingModel）。

创建 WCF 服务

现在是时候添加 WCF 服务来公开数据，接下来我们将创建该项目。

- 文件->添加>项目..。
- 从左窗格中选择“Visual C#”，然后选择“WCF 服务应用程序”
- 输入 STESample 作为名称，然后单击“确定”
- 添加对 system.web 程序集的引用

- 添加对STESample和STESample项目的引用

需要将 EF 连接字符串复制到此项目中，以便在运行时找到它。

- 打开 STESample 项目的app.config文件并复制connectionStrings元素
- 在STESample项目中，将connectionStrings元素粘贴为web.config文件的configuration元素的子元素

现在是时候实现实际服务了。

- 打开IService1.cs并将内容替换为以下代码

```
using System.Collections.Generic;
using System.ServiceModel;

namespace STESample.Service
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        List<Blog> GetBlogs();

        [OperationContract]
        void UpdateBlog(Blog blog);
    }
}
```

- 打开Service1并将内容替换为以下代码

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;

namespace STESample.Service
{
    public class Service1 : IService1
    {
        /// <summary>
        /// Gets all the Blogs and related Posts.
        /// </summary>
        public List<Blog> GetBlogs()
        {
            using (BloggingContext context = new BloggingContext())
            {
                return context.Blogs.Include("Posts").ToList();
            }
        }

        /// <summary>
        /// Updates Blog and its related Posts.
        /// </summary>
        public void UpdateBlog(Blog blog)
        {
            using (BloggingContext context = new BloggingContext())
            {
                try
                {
                    // TODO: Perform validation on the updated order before applying the changes.

                    // The ApplyChanges method examines the change tracking information
                    // contained in the graph of self-tracking entities to infer the set of operations
                    // that need to be performed to reflect the changes in the database.
                    context.Blogs.ApplyChanges(blog);
                    context.SaveChanges();

                }
                catch (UpdateException)
                {
                    // To avoid propagating exception messages that contain sensitive data to the client
                    tier
                    // calls to ApplyChanges and SaveChanges should be wrapped in exception handling code.
                    throw new InvalidOperationException("Failed to update. Try your request again.");
                }
            }
        }
    }
}

```

从控制台应用程序使用服务

让我们创建一个使用我们的服务的控制台应用程序。

- 文件->> 项目 ..。
- 从左窗格中选择 "Visual C#"，然后选择"控制台应用程序"
- 输入STESample作为名称，然后单击 "确定"
- 添加对STESample项目的引用

我们需要对 WCF 服务的服务引用

- 在解决方案资源管理器中右键单击 "ConsoleTest" 项目，然后选择 "添加服务引用 ..."
- 单击发现

- 输入BloggingService作为命名空间，然后单击“确定”

现在，我们可以编写一些代码来使用该服务。

- 打开Program.cs并将内容替换为以下代码。

```
using STESample.ConsoleTest.BloggingService;
using System;
using System.Linq;

namespace STESample.ConsoleTest
{
    class Program
    {
        static void Main(string[] args)
        {
            // Print out the data before we change anything
            Console.WriteLine("Initial Data:");
            DisplayBlogsAndPosts();

            // Add a new Blog and some Posts
            AddBlogAndPost();
            Console.WriteLine("After Adding:");
            DisplayBlogsAndPosts();

            // Modify the Blog and one of its Posts
            UpdateBlogAndPost();
            Console.WriteLine("After Update:");
            DisplayBlogsAndPosts();

            // Delete the Blog and its Posts
            DeleteBlogAndPost();
            Console.WriteLine("After Delete:");
            DisplayBlogsAndPosts();

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }

        static void DisplayBlogsAndPosts()
        {
            using (var service = new Service1Client())
            {
                // Get all Blogs (and Posts) from the service
                // and print them to the console
                var blogs = service.GetBlogs();
                foreach (var blog in blogs)
                {
                    Console.WriteLine(blog.Name);
                    foreach (var post in blog.Posts)
                    {
                        Console.WriteLine(" - {0}", post.Title);
                    }
                }
            }

            Console.WriteLine();
            Console.WriteLine();
        }

        static void AddBlogAndPost()
        {
            using (var service = new Service1Client())
            {
                // Create a new Blog with a couple of Posts
                var newBlog = new Blog
                {
                    Name = "The .NET Guru"
                };
                service.AddBlog(newBlog);

                var newPost = new Post
                {
                    Title = "Introduction to WCF Services"
                };
                service.AddPost(newPost);
            }
        }
    }
}
```

```

        Name = "The New Blog",
        Posts =
        {
            new Post { Title = "Welcome to the new blog"},
            new Post { Title = "What's new on the new blog"}
        }
    };

    // Save the changes using the service
    service.UpdateBlog(newBlog);
}
}

static void UpdateBlogAndPost()
{
    using (var service = new Service1Client())
    {
        // Get all the Blogs
        var blogs = service.GetBlogs();

        // Use LINQ to Objects to find The New Blog
        var blog = blogs.First(b => b.Name == "The New Blog");

        // Update the Blogs name
        blog.Name = "The Not-So-New Blog";

        // Update one of the related posts
        blog.Posts.First().Content = "Some interesting content...";

        // Save the changes using the service
        service.UpdateBlog(blog);
    }
}

static void DeleteBlogAndPost()
{
    using (var service = new Service1Client())
    {
        // Get all the Blogs
        var blogs = service.GetBlogs();

        // Use LINQ to Objects to find The Not-So-New Blog
        var blog = blogs.First(b => b.Name == "The Not-So-New Blog");

        // Mark all related Posts for deletion
        // We need to call ToList because each Post will be removed from the
        // Posts collection when we call MarkAsDeleted
        foreach (var post in blog.Posts.ToList())
        {
            post.MarkAsDeleted();
        }

        // Mark the Blog for deletion
        blog.MarkAsDeleted();

        // Save the changes using the service
        service.UpdateBlog(blog);
    }
}
}
}

```

现在可以运行应用程序来查看其实际运行情况。

- 在解决方案资源管理器中右键单击 "ConsoleTest" 项目，然后选择 "调试-> 启动新实例"

当应用程序执行时，将看到以下输出。

Initial Data:

ADO.NET Blog

- Intro to EF

- What is New

After Adding:

ADO.NET Blog

- Intro to EF

- What is New

The New Blog

- Welcome to the new blog

- What's new on the new blog

After Update:

ADO.NET Blog

- Intro to EF

- What is New

The Not-So-New Blog

- Welcome to the new blog

- What's new on the new blog

After Delete:

ADO.NET Blog

- Intro to EF

- What is New

Press any key to exit...

从 WPF 应用程序使用服务

让我们创建一个使用我们的服务的 WPF 应用程序。

- 文件->> 项目 ..。
- 从左窗格中选择 "Visual C#"，然后选择" WPF 应用程序"
- 输入**STESample**作为名称, 然后单击 "确定"
- 添加对**STESample**项目的引用

我们需要对 WCF 服务的服务引用

- 在解决方案资源管理器中右键单击 "WPFTest" 项目, 然后选择 "添加服务引用 ..."
- 单击发现
- 输入**BloggingService**作为命名空间, 然后单击 "确定"

现在, 我们可以编写一些代码来使用该服务。

- 打开**mainwindow.xaml**并将内容替换为以下代码。

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:STESample="clr-namespace:STESample;assembly=STESample.Entities"
    mc:Ignorable="d" x:Class="STESample.WPFTest.MainWindow"
    Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded">

    <Window.Resources>
        <CollectionViewSource
            x:Key="blogViewSource"
            d:DesignSource="{d:DesignInstance {x:Type STESample:Blog}, CreateList=True}"/>
        <CollectionViewSource
            x:Key="blogPostsViewSource"
            Source="{Binding Posts, Source={StaticResource blogViewSource}}"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource blogViewSource}">
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding}" Margin="10,10,10,179">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding BlogId}" Header="Id" Width="Auto" IsReadOnly="True"
/>
                <DataGridTextColumn Binding="{Binding Name}" Header="Name" Width="Auto"/>
                <DataGridTextColumn Binding="{Binding Url}" Header="Url" Width="Auto"/>
            </DataGrid.Columns>
        </DataGrid>
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding Source={StaticResource blogPostsViewSource}}"
Margin="10,145,10,38">
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding PostId}" Header="Id" Width="Auto"
IsReadOnly="True"/>
                <DataGridTextColumn Binding="{Binding Title}" Header="Title" Width="Auto"/>
                <DataGridTextColumn Binding="{Binding Content}" Header="Content" Width="Auto"/>
            </DataGrid.Columns>
        </DataGrid>
        <Button Width="68" Height="23" HorizontalAlignment="Right" VerticalAlignment="Bottom"
Margin="0,0,10,10" Click="buttonSave_Click">Save</Button>
    </Grid>
</Window>

```

- 打开 Mainwindow.xaml (MainWindow.xaml.cs) 的隐藏代码，并将内容替换为以下代码

```

using STESample.WPFTest.BloggingService;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Data;

namespace STESample.WPFTest
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            using (var service = new Service1Client())
            {
                // Find the view source for Blogs and populate it with all Blogs (and related Posts)
                // from the Service. The default editing functionality of WPF will allow the objects
                // to be manipulated on the screen.
                var blogsViewSource = (CollectionViewSource)this.FindResource("blogViewSource");
                blogsViewSource.Source = service.GetBlogs().ToList();
            }
        }

        private void buttonSave_Click(object sender, RoutedEventArgs e)
        {
            using (var service = new Service1Client())
            {
                // Get the blogs that are bound to the screen
                var blogsViewSource = (CollectionViewSource)this.FindResource("blogViewSource");
                var blogs = (List<Blog>)blogsViewSource.Source;

                // Save all Blogs and related Posts
                foreach (var blog in blogs)
                {
                    service.UpdateBlog(blog);
                }

                // Re-query for data to get database-generated keys etc.
                blogsViewSource.Source = service.GetBlogs().ToList();
            }
        }
    }
}

```

现在可以运行应用程序来查看其实际运行情况。

- 在解决方案资源管理器中右键单击 "WPFTest" 项目，然后选择 "调试-> 启动新实例"
- 你可以使用屏幕操作数据，并使用 "保存" 按钮通过服务保存数据

Blog Id	Name	Url
1	ADO.NET Blog	blogs.msdn.com/adonet
3	My Blog	

Post Id	Title	Content
5	Welcome	This is my first post...

记录和截取数据库操作

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

从实体框架6开始，无论何时实体框架将命令发送到数据库，都可以通过应用程序代码截获此命令。这通常用于记录 SQL，但也可用于修改或中止命令。

具体而言，EF 包括：

- 与 `DataContext` 类似的上下文的日志属性。登录 LINQ to SQL
- 用于自定义发送到日志的输出内容和格式的机制
- 用于侦听的低级别构建块，提供更好的控制/灵活性

Context 日志属性

可以将 `DbContext` 属性设置为采用字符串的任何方法的委托。最常见的情况是，通过将其设置为该该设置的 "Write" 方法，将其与任何未任何类型一起使用。当前上下文生成的所有 SQL 都将记录到该编写器。例如，以下代码将 SQL 日志记录到控制台：

```
using (var context = new BlogContext())
{
    context.Database.Log = Console.WriteLine;

    // Your code here...
}
```

请注意，上下文。数据库。日志设置为 `Console`。这是将 SQL 日志记录到控制台所需的全部。

让我们添加一些简单的查询/插入/更新代码，以便我们可以看到一些输出：

```
using (var context = new BlogContext())
{
    context.Database.Log = Console.WriteLine;

    var blog = context.Blogs.First(b => b.Title == "One Unicorn");

    blog.Posts.First().Title = "Green Eggs and Ham";

    blog.Posts.Add(new Post { Title = "I do not like them!" });

    context.SaveChangesAsync().Wait();
}
```

这将生成以下输出：

```

SELECT TOP (1)
    [Extent1].[Id] AS [Id],
    [Extent1].[Title] AS [Title]
FROM [dbo].[Blogs] AS [Extent1]
WHERE (N'One Unicorn' = [Extent1].[Title]) AND ([Extent1].[Title] IS NOT NULL)
-- Executing at 10/8/2013 10:55:41 AM -07:00
-- Completed in 4 ms with result: SqlDataReader

SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Title] AS [Title],
    [Extent1].[BlogId] AS [BlogId]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[BlogId] = @EntityKeyValue1
-- EntityKeyValue1: '1' (Type = Int32)
-- Executing at 10/8/2013 10:55:41 AM -07:00
-- Completed in 2 ms with result: SqlDataReader

UPDATE [dbo].[Posts]
SET [Title] = @0
WHERE ([Id] = @1)
-- @0: 'Green Eggs and Ham' (Type = String, Size = -1)
-- @1: '1' (Type = Int32)
-- Executing asynchronously at 10/8/2013 10:55:41 AM -07:00
-- Completed in 12 ms with result: 1

INSERT [dbo].[Posts]([Title], [BlogId])
VALUES (@0, @1)
SELECT [Id]
FROM [dbo].[Posts]
WHERE @@ROWCOUNT > 0 AND [Id] = scope_identity()
-- @0: 'I do not like them!' (Type = String, Size = -1)
-- @1: '1' (Type = Int32)
-- Executing asynchronously at 10/8/2013 10:55:41 AM -07:00
-- Completed in 2 ms with result: SqlDataReader

```

(请注意，这是假设已发生任何数据库初始化的输出。如果尚未发生数据库初始化，则会有更多的输出，显示所有工作迁移在这些情况下执行的操作，以检查或创建新数据库。)

记录了哪些内容？

设置日志属性后，将记录以下所有内容：

- 适用于所有不同类型的命令的 SQL。例如：
 - 查询，包括常规 LINQ 查询、eSQL 查询和来自 `SqlQuery` 等方法的原始查询
 - 作为 `SaveChanges` 的一部分生成的插入、更新和删除操作
 - 关系加载查询，如延迟加载生成的查询
- 参数
- 命令是否正在以异步方式执行
- 指示命令开始执行的时间的时间戳
- 命令是否已成功完成，引发异常，或是否已取消（对于异步）
- 结果值的一些指示
- 执行命令所花费的大致时间。请注意，这是发送命令以返回结果对象的时间。它不包含读取结果的时间。

查看上面的示例输出，记录的四个命令中的每一个都是：

- 通过调用上下文产生的查询。博客。首先
 - 请注意，获取 SQL 的 `ToString` 方法不能处理此查询，因为 "First" 不提供可调用 `ToString` 的 `IQueryable`
- 由博客的延迟加载引起的查询。创纪录

- 请注意延迟加载发生的键值的参数详细信息
- 仅记录设置为非默认值的参数的属性。例如，仅当 Size 属性为非零时，才会显示该属性。
- SaveChangesAsync 生成的两个命令;一项用于更改发布标题的更新，另一种用于添加新的帖子
 - 请注意 FK 和 Title 属性的参数详细信息
 - 请注意，这些命令是异步执行的

记录到不同的位置

如上所示，记录到控制台非常简单。通过使用不同类型的类型，也可以很容易地记录到内存、文件等。

如果你熟悉 LINQ to SQL 你可能会注意到，在 LINQ to SQL 日志属性设置为实际的“设置”对象（例如，在“控制台”中），而在 EF 中，Log 属性被设置为接受字符串的方法（例如，Console.WriteLine 或 Console.WriteLine）。这种情况的原因是，通过接受可充当字符串接收器的任何委托，将 EF 从无效中分离。例如，假设已有一些日志记录框架，并定义如下所示的日志记录方法：

```
public class MyLogger
{
    public void Log(string component, string message)
    {
        Console.WriteLine("Component: {0} Message: {1}", component, message);
    }
}
```

此操作可能会挂接到 EF 日志属性，如下所示：

```
var logger = new MyLogger();
context.Database.Log = s => logger.Log("EFApp", s);
```

结果日志记录

默认情况下，在将命令发送到数据库之前，将使用时间戳来记录命令文本（SQL）、参数和“执行”行。执行命令后，将记录包含运行时间的“已完成”行。

请注意，对于异步命令，在异步任务实际完成、失败或取消之前，不会记录“已完成”行。

“已完成”行包含不同的信息，具体取决于命令的类型以及执行是否成功。

成功执行

对于成功完成的命令，输出为“在 x 毫秒内完成，结果为：”，后跟一些对结果的指示。对于返回数据读取器的命令，结果指示是返回的DbDataReader的类型。对于返回整数值的命令，如显示的结果上方显示的 update 命令，则为该整数。

执行失败

对于通过引发异常而失败的命令，输出包含来自异常的消息。例如，使用SqlQuery 对存在的表进行查询时，会生成如下所示的日志输出：

```
SELECT * from ThisTableIsMissing
-- Executing at 5/13/2013 10:19:05 AM
-- Failed in 1 ms with error: Invalid object name 'ThisTableIsMissing'.
```

已取消执行

对于取消任务的异步命令，结果可能会失败，并出现异常，因为这是当尝试取消时，基础 ADO.NET 提供程序经常执行的操作。如果这种情况不会发生并且任务完全取消，则输出将如下所示：

```
update Blogs set Title = 'No' where Id = -1
-- Executing asynchronously at 5/13/2013 10:21:10 AM
-- Canceled in 1 ms
```

更改日志内容和格式

在“数据库概述”下，使用 `DatabaseLogFormatter` 对象。此对象有效地将 `IDbCommandInterceptor` 实现（见下文）绑定到接受字符串和 `DbContext` 的委托。这意味着，`DatabaseLogFormatter` 上的方法是在执行命令之前和之后调用的。这些 `DatabaseLogFormatter` 方法收集并格式化日志输出，并将其发送到委托。

自定义 `DatabaseLogFormatter`

可以通过创建从 `DatabaseLogFormatter` 派生的新类并根据需要重写方法，从而更改所记录的内容和格式。要重写的最常见方法是：

- `LogCommand` – 重写此方法以更改命令在执行之前的记录方式。默认情况下，`LogCommand` 为每个参数调用 `LogParameter`；您可以选择在重写中执行相同的操作，或以不同的方式处理参数。
- `LogResult` – 重写此方法以更改如何记录执行命令的结果。
- `LogParameter` – 重写此参数以更改参数日志记录的格式和内容。

例如，假设我们想要在每个命令发送到数据库之前只记录一行。可以通过两个替代来完成此操作：

- 重写 `LogCommand` 以格式化和写入单一的 SQL 行
- 重写 `LogResult` 不执行任何操作。

代码如下所示：

```
public class OneLineFormatter : DatabaseLogFormatter
{
    public OneLineFormatter(DbContext context, Action<string> writeAction)
        : base(context, writeAction)
    {
    }

    public override void LogCommand<TResult>(
        DbCommand command, DbContextInterceptionContext<TResult> interceptionContext)
    {
        Write(string.Format(
            "Context '{0}' is executing command '{1}' '{2}'",
            Context.GetType().Name,
            command.CommandText.Replace(Environment.NewLine, ""),
            Environment.NewLine));
    }

    public override void LogResult<TResult>(
        DbCommand command, DbContextInterceptionContext<TResult> interceptionContext)
    {
    }
}
```

若要日志输出，只需调用写入方法，该方法会将输出发送到配置的写入委托。

（请注意，此代码只是为了举例地删除换行符。它在查看复杂的 SQL 时可能无法正常工作。）

设置 `DatabaseLogFormatter`

创建新的 `DatabaseLogFormatter` 类后，需要将其注册到 EF。这是使用基于代码的配置来完成的。简而言之，这意味着要在与 `DbContext` 类相同的程序集中创建一个从 `DbConfiguration` 派生的新类，然后在此新类的构造函数中调用 `SetDatabaseLogFormatter`。例如：

```
public class MyDbConfiguration : DbConfiguration
{
    public MyDbConfiguration()
    {
        SetDatabaseLogFormatter(
            (context, writeAction) => new OneLineFormatter(context, writeAction));
    }
}
```

使用新的 DatabaseLogFormatter

此新 DatabaseLogFormatter 将在每次设置数据库时使用。因此，运行第1部分中的代码将生成以下输出：

```
Context 'BlogContext' is executing command 'SELECT TOP (1) [Extent1].[Id] AS [Id], [Extent1].[Title] AS [Title]FROM [dbo].[Blogs] AS [Extent1]WHERE (N'One Unicorn' = [Extent1].[Title]) AND ([Extent1].[Title] IS NOT NULL)'
Context 'BlogContext' is executing command 'SELECT [Extent1].[Id] AS [Id], [Extent1].[Title] AS [Title], [Extent1].[BlogId] AS [BlogId]FROM [dbo].[Posts] AS [Extent1]WHERE [Extent1].[BlogId] = @EntityKeyValue1'
Context 'BlogContext' is executing command 'update [dbo].[Posts]set [Title] = @0where ([Id] = @1)'
Context 'BlogContext' is executing command 'insert [dbo].[Posts]([Title], [BlogId])values (@0, @1)select [Id]from [dbo].[Posts]where @@rowcount > 0 and [Id] = scope_identity()'
```

拦截构建块

到目前为止，我们已经介绍了如何使用 DbContext 记录由 EF 生成的 SQL。但在某些低级别构建块上，此代码实际上是相对较窄的外观，用于更常见的侦听。

侦听接口

拦截代码是围绕截取接口的概念构建的。这些接口从 IDbInterceptor 继承，并定义在 EF 执行某个操作时调用的方法。目的是要截获每种类型的对象的一个接口。例如， IDbCommandInterceptor 接口定义在 EF 调用 ExecuteNonQuery、ExecuteScalar、ExecuteReader 和相关方法之前调用的方法。同样，接口会定义在这些操作完成时调用的方法。以上所示的 DatabaseLogFormatter 类实现此接口来记录命令。

截获上下文

查看在任何侦听器接口上定义的方法很明显，就是每个调用都给定 DbInterceptionContext 类型的对象或派生的某种类型，如 DbCommandInterceptionContext<>。此对象包含有关 EF 正在采取的操作的上下文信息。例如，如果该操作是代表 DbContext 执行的，则 DbContext 将包含在 DbInterceptionContext 中。同样，对于异步执行的命令，将在 DbCommandInterceptionContext 上设置 IsAsync 标志。

结果处理

DbCommandInterceptionContext<> 类包含一个名为 Result、OriginalResult、Exception 和 OriginalException 的属性。对于对在执行操作之前调用的截取方法的调用，这些属性将设置为 null/零（即，对于正在执行方法。如果执行并成功执行该操作，则结果和 OriginalResult 将设置为操作的结果。然后，可以在执行操作后调用的截取方法中观察这些值，即已执行方法。同样，如果操作引发，则会设置 Exception 和 OriginalException 属性。

禁止执行

如果侦听器在执行命令之前设置 Result 属性（在其中一个执行方法）后，EF 不会尝试实际执行该命令，而只是使用结果集。换句话说，侦听器可以取消命令的执行，但会继续执行 EF，就像执行命令一样。

通常使用包装提供程序执行的命令批处理就是使用此方法的示例。侦听器将以批处理的形式存储命令以供以后执行，但会“假设”命令已正常执行。请注意，它需要超过此数目才能实现批处理，但这是如何使用更改截取结果的示例。

还可以通过在其中一个“...”正在执行方法。这会导致 EF 通过引发给定异常来继续执行操作。当然，这可能会导致应用程序崩溃，但也可能是由 EF 处理的暂时性异常或其他异常。例如，在命令执行失败时，可以在测试环境中使用它来测试应用程序的行为。

在执行后更改结果

如果侦听器在执行命令后设置 Result 属性(在其中一个 .Execute 方法)后, EF 将使用更改后的结果, 而不是实际从操作返回的结果。同样, 如果侦听器在执行命令后设置了 Exception 属性, 则 EF 将引发设置异常, 就好像操作引发了异常一样。

侦听器还可以将 Exception 属性设置为 null, 以指示不应引发异常。如果执行操作失败, 则此方法会很有用, 但侦听器希望 EF 继续操作, 就像操作已成功一样。这通常还涉及到设置结果, 以便 EF 有一些结果值在继续时使用。

OriginalResult 和 OriginalException

在 EF 执行了某一操作后, 如果执行未通过, 则将设置 Result 和 OriginalResult 属性, 如果执行失败但出现异常, 则将设置为 Exception 和 OriginalException 属性。

OriginalResult 和 OriginalException 属性是只读的, 并且仅在实际执行操作后由 EF 设置。侦听器不能设置这些属性。这意味着, 任何侦听器都可以区分已由其他侦听器设置的异常或结果, 而不是执行操作时所发生的真实异常或结果。

正在注册侦听器

一旦创建了一个或多个截获接口的类, 就可以使用 DbInterception 类向 EF 注册它。例如:

```
DbInterception.Add(new NLogCommandInterceptor());
```

还可以使用基于 DbConfiguration 代码的配置机制在应用域级别注册拦截程序。

示例: 记录到 NLog

让我们将这一切结合起来, 使用 IDbCommandInterceptor 和 NLog 来执行以下操作:

- 为非异步执行的任何命令记录警告
- 为执行时引发的任何命令记录错误

下面是执行日志记录的类, 应如下所示进行注册:

```
public class NLogCommandInterceptor : IDbCommandInterceptor
{
    private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

    public void NonQueryExecuting(
        SqlCommand command, SqlCommandInterceptionContext<int> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void NonQueryExecuted(
        SqlCommand command, SqlCommandInterceptionContext<int> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ReaderExecuting(
        SqlCommand command, SqlCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ReaderExecuted(
        SqlCommand command, SqlCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    public void ScalarExecuting(
        SqlCommand command, SqlCommandInterceptionContext<object> interceptionContext)
    {
        LogIfNonAsync(command, interceptionContext);
    }

    public void ScalarExecuted(
        SqlCommand command, SqlCommandInterceptionContext<object> interceptionContext)
    {
        LogIfError(command, interceptionContext);
    }

    private void LogIfNonAsync<TResult>(
        SqlCommand command, SqlCommandInterceptionContext<TResult> interceptionContext)
    {
        if (!interceptionContext.IsAsync)
        {
            Logger.Warn("Non-async command used: {0}", command.CommandText);
        }
    }

    private void LogIfError<TResult>(
        SqlCommand command, SqlCommandInterceptionContext<TResult> interceptionContext)
    {
        if (interceptionContext.Exception != null)
        {
            Logger.Error("Command {0} failed with exception {1}",
                command.CommandText, interceptionContext.Exception);
        }
    }
}
```

请注意，此代码如何使用侦听上下文来发现何时以非异步方式执行命令，并在执行命令时发现错误。

EF 4、5和6的性能注意事项

2020/3/11 ·

按 David Obando、Eric Dettinger 和其他

发布日期:2012年4月

上次更新时间:可能为2014

1. 介绍

对象关系映射框架是一种简便的方法，可用于在面向对象的应用程序中提供数据访问的抽象。对于 .NET 应用程序，Microsoft 推荐的 O/RM 实体框架。但对于任何抽象，性能都可能会成为问题。

本文的目的是为了演示使用实体框架开发应用程序时的性能注意事项，使开发人员了解可能会影响性能的实体框架内部算法，并提供调查和提高使用实体框架的应用程序的性能。Web 上已经提供了有关性能的很多好主题，还尝试尽可能指向这些资源。

性能是一个棘手的话题。本白皮书旨在提供一种资源，帮助你为使用实体框架的应用程序做出性能相关的决策。我们提供了一些测试指标来演示性能，但这些度量值并不是要在应用程序中看到的性能的绝对指标。

出于实用的目的，本文档假设在 .net 4.0 下运行实体框架4，实体框架5和6在 .NET 4.5 下运行。为实体框架5所做的许多性能改进都位于 .NET 4.5 随附的核心组件中。

实体框架6是带外版本，不依赖于随 .NET 提供实体框架组件。实体框架6同时适用于 .NET 4.0 和 .NET 4.5，并可为尚未从 .NET 4.0 升级但希望应用程序中的最新实体框架位的用户提供大性能优势。当本文档提到实体框架6时，它是指在撰写本文时可用的最新版本：版本6.1.0。

2. 冷查询和热查询执行

第一次对给定模型进行任何查询时，实体框架在幕后执行大量工作以加载和验证模型。我们经常将此第一个查询称为 "冷" 查询。针对已加载模型的进一步查询称为 "热" 查询，并更快。

让我们大致了解一下使用实体框架执行查询时所用的时间，并查看在实体框架6中的改进情况。

第一次查询执行-冷查询

模型	操作	EF4 性能	EF5 性能	EF6 性能
<pre>using(var db = new MyContext()) {</pre>	上下文创建	中等	中等	低
<pre>var q1 = from c in db.Customers where c.Id == id1 select c;</pre>	查询表达式创建	低	低	低

		EF4	EF5	EF6
<pre>var c1 = q1.First();</pre>	LINQ 查询执行	<ul style="list-style-type: none"> -元数据加载:高但已缓存 -视图生成:可能非常高但已被缓存 -参数求值:中型 -查询转换:中型 -Materializer 生成:中型但已缓存 -数据库查询执行:可能很高 <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型 -Identity lookup:中型 	<ul style="list-style-type: none"> -元数据加载:高但已缓存 -视图生成:可能非常高但已被缓存 -参数求值:低 -查询转换:中型但已缓存 -Materializer 生成:中型但已缓存 -数据库查询执行:可能很高(某些情况下更好的查询) <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型 -Identity lookup:中型 	<ul style="list-style-type: none"> -元数据加载:高但已缓存 -视图生成:中型但已缓存 -参数求值:低 -查询转换:中型但已缓存 -Materializer 生成:中型但已缓存 -数据库查询执行:可能很高(某些情况下更好的查询) <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型(速度快于 EF5) -Identity lookup:中型
}	Connection.Close	低	低	低

第二个查询执行–热查询

		EF4	EF5	EF6
<pre>using(var db = new MyContext()) {</pre>	上下文创建	中等	中等	低
<pre>var q1 = from c in db.Customers where c.Id == id1 select c;</pre>	查询表达式创建	低	低	低
<pre>var c1 = q1.First();</pre>	LINQ 查询执行	<ul style="list-style-type: none"> -元数据加载查找:高但高速缓存低 -查看生成查找:可能非常高但缓存较低 -参数求值:中型 -查询转换查找:中型 -Materializer生成查找:中速但缓存低 -数据库查询执行:可能很高 <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型 -Identity lookup:中型 	<ul style="list-style-type: none"> -元数据加载查找:高但高速缓存低 -查看生成查找:可能非常高但缓存较低 -参数求值:低 -查询转换查找:中但缓存低 -Materializer生成查找:中速但缓存低 -数据库查询执行:可能很高(某些情况下更好的查询) <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型 -Identity lookup:中型 	<ul style="list-style-type: none"> -元数据加载查找:高但高速缓存低 -查看生成查找:中但缓存低 -参数求值:低 -查询转换查找:中但缓存低 -Materializer生成查找:中速但缓存低 -数据库查询执行:可能很高(某些情况下更好的查询) <ul style="list-style-type: none"> + 连接打开 + ExecuteReader + DataReader。读取对象具体化:中型(速度快于 EF5) -Identity lookup:中型
}	Connection.Close	低	低	低

可以通过多种方式降低冷查询和热查询的性能成本，我们将在下一节中介绍这些方法。具体而言，我们将通过使用预生成的视图来了解如何通过使用预生成的视图来减少冷查询中模型加载的成本，这有助于减少在视图生成

过程中遇到的性能难题。对于热查询，我们将介绍查询计划缓存，无跟踪查询，以及不同的查询执行选项。

2.1 什么是视图生成？

为了了解生成的视图，我们必须首先了解“映射视图”是什么。映射视图是映射中为每个实体集和关联指定的转换的可执行表示形式。在内部，这些映射视图采用 CQTs（规范查询树）的形式。有两种类型的映射视图：

- 查询视图：这表示从数据库架构转到概念模型所需的转换。
- 更新视图：这表示从概念模型转换到数据库架构所需的转换。

请记住，概念模型可能不同于数据库架构。例如，一个表可能用于存储两个不同实体类型的数据。继承和非日常映射在映射视图的复杂性方面扮演着角色。

基于映射规范计算这些视图的过程是所谓的视图生成。在加载模型时，或在生成时使用“预生成的视图”，可以动态地执行视图生成；后者以实体 SQL 语句的形式序列化为 C# 或 VB 文件。

当生成视图时，还会对其进行验证。从性能角度来看，视图生成的大部分开销实际上是对视图的验证，这可以确保实体之间的连接有意义，并且具有所有支持操作的正确基数。

当执行对实体集的查询时，查询将与相应的查询视图结合使用，此组合的结果将通过计划编译器运行，以创建后备存储可以理解的查询表示形式。对于 SQL Server，此编译的最终结果将为 T-SQL SELECT 语句。第一次执行对实体集的更新时，更新视图将通过类似的进程运行，以将其转换为目标数据库的 DML 语句。

影响视图生成性能的2.2 因素

视图生成步骤的性能不仅取决于模型的大小，还取决于模型的互连方式。如果通过继承链或关联来连接两个实体，则称它们处于连接状态。同样，如果两个表通过外键连接，它们就会连接起来。随着架构中连接的实体和表的数量增加，视图生成成本会增加。

在最糟糕的情况下，用于生成和验证视图的算法是指数的，尽管我们确实使用某些优化来改进此功能。对性能产生负面影响的最大因素包括：

- 模型大小，引用实体数以及这些实体之间的关联量。
- 模型复杂性，尤其是涉及大量类型的继承。
- 使用独立关联，而不是外键关联。

对于小型的简单模型，成本可能会足够小，无法使用预先生成的视图。随着模型大小和复杂性的增加，可以使用多个选项来降低查看生成和验证的成本。

2.3 使用预生成的视图缩短模型加载时间

有关如何使用实体框架 6 上预生成的视图的详细信息，请访问[预生成的映射视图](#)

使用实体框架 Power Tools 社区版的2.3.1 预生成的视图

您可以使用[实体框架 6 Power Tools 社区版](#)通过右键单击模型类文件并使用实体框架菜单选择“生成视图”来生成 EDMX 和 Code First 模型的视图。实体框架 Power Tools 社区版仅适用于 DbContext 派生的上下文。

2.3.2 如何通过 Edmgen.exe 创建的模型使用预生成的视图

Edmgen.exe 是随 .NET 提供的实用程序，与实体框架 4 和 5（而不是实体框架 6）结合使用。Edmgen.exe 允许您从命令行生成模型文件、对象层和视图。其中一个输出将是您选择的语言（VB 或 C#）的视图文件。这是一个代码文件，其中包含每个实体集的实体 SQL 代码段。若要启用预生成的视图，只需将该文件包含在项目中。

如果手动编辑模型的架构文件，则需要重新生成视图文件。可以通过使用 /mode: ViewGeneration 标志运行 edmgen.exe 来实现此目的。

2.3.3 如何使用带有 EDMX 文件的预生成视图

你还可以使用 Edmgen.exe 为 EDMX 文件生成视图—前面引用的 MSDN 主题介绍如何添加预生成事件来执行此操作，但这很复杂，但在某些情况下，不可能。通常，在模型位于 edmx 文件中时，使用 T4 模板生成视图通常更容易。

ADO.NET 团队博客提供了一篇文章，介绍如何使用 T4 模板生成视图

(<http://blogs.msdn.com/b/adonet/archive/2008/06/20/how-to-use-a-t4-template-for-view-generation.aspx>)。

此文章包含可下载并添加到项目中的模板。模板是为实体框架的第一个版本而编写的，因此不能保证它们使用最新版本的实体框架。不过，你可以为实体框架4和5从 Visual Studio 库下载更多最新的视图生成模板集：

- VB.NET: <<http://visualstudiogallery.msdn.microsoft.com/118b44f2-1b91-4de2-a584-7a680418941d>>
- C#: <<http://visualstudiogallery.msdn.microsoft.com/ae7730ce-ddab-470f-8456-1b313cd2c44d>>

如果使用实体框架6，则可以从 Visual Studio 库中的 <<http://visualstudiogallery.msdn.microsoft.com/18a7db90-6705-4d19-9dd1-0a6c23d0751f>> 获取查看生成 T4 模板。

2.4 降低视图生成成本

使用预生成的视图会将视图生成的成本从模型加载(运行时)移动到设计时。虽然这在运行时提高了启动性能，但在开发时仍会遇到视图生成难点。在编译时和运行时，有几个其他技巧可以帮助降低视图生成的成本。

2.4.1 使用外键关联降低视图生成成本

我们发现了很多情况，即从独立的关联关联到外键关联，将模型中的关联切换到外键关联大大缩短了视图生成所花费的时间。

为了演示这种改进，我们使用 Edmgen.exe 生成了两个版本的 Navision 模型。*注意：有关 Navision 模型的说明，请参阅附录 C。* 由于在此练习中，Navision 模型的实体和关系非常多，因此它很感兴趣。

此非常大的模型的一个版本是通过外键关联生成的，另一个版本是通过独立关联生成的。接下来，我们将为每个模型生成视图所需的时间。实体框架5测试使用 EntityViewGenerator 类中的 GenerateViews() 方法来生成视图，而实体框架6测试使用类 StorageMappingItemCollection 中的 GenerateViews() 方法。这是因为实体框架6代码库中发生了代码重构。

使用实体框架5，在实验室计算机中使用外键查看模型的生成耗时为65分钟。未知的是为使用独立关联的模型生成视图所需的时间。在实验室中，我们将运行的测试留给了在实验室中重新启动计算机以安装每月更新。

使用实体框架6，在同一实验室计算机中使用外键查看模型的生成时间为28秒。使用独立关联的模型的视图生成花费了58秒。对视图生成代码实体框架6进行的改进意味着许多项目不需要预生成的视图即可获得更快的启动时间。

请注意，在实体框架4和5中预生成的视图可以通过 Edmgen.exe 或实体框架的增强工具来完成。对于实体框架6视图，可以通过实体框架的 Power Tools 或编程方式完成，如[预生成的映射视图](#)中所述。

2.4.1.1 如何使用外键，而不是独立关联

在 Visual Studio 中使用 Edmgen.exe 或 Entity Designer 时，默认情况下会获取 Fk，并只使用单个 checkbox 或命令行标志在 Fk 和 IAs 之间切换。

如果使用的是大型 Code First 模型，则使用独立关联将对视图生成产生相同的效果。可以通过在依赖对象的类上包含外键属性来避免这种影响，尽管某些开发人员会将其视为污染其对象模型。可以在

<<http://blog.oneunicorn.com/2011/12/11/whats-the-deal-with-mapping-foreign-keys-using-the-entity-framework/>> 中找到有关此主题的详细信息。

实体设计器	Edmgen.exe	Code First
实体设计器	在两个实体之间添加关联后，请确保具有引用约束。引用约束告诉实体框架使用外键，而不是独立关联。有关更多详细信息，请访问< http://blogs.msdn.com/b/efdesign/archive/2009/03/16/foreign-keys-in-the-entity-framework.aspx >。	
Edmgen.exe	当使用 Edmgen.exe 从数据库生成文件时，将遵循外键，并将其添加到模型中。有关 Edmgen.exe 公开的不同选项的详细信息，请 http://msdn.microsoft.com/library/bb387165.aspx 。	
Code First	有关如何在使用 Code First 时包含依赖对象的外键属性的信息，请参阅 Code First 约定 主题的“关系约定”部分。	

2.4.2 sections 将模型移到单独的程序集

如果您的模型直接包含在您的应用程序的项目中，并通过预生成事件或 T4 模板生成视图，则在重新生成项目时，将会进行查看生成和验证，即使模型未更改也是如此。如果将模型移到单独的程序集，并从应用程序的项目中引用它，则可以对应用程序进行其他更改，而无需重新生成包含该模型的项目。

注意：将模型移动到不同的程序集时，请记住将模型的连接字符串复制到客户端项目的应用程序配置文件中。

2.4.3 禁用基于 edmx 的模型的验证

在编译时验证 EDMX 模型，即使模型未更改也是如此。如果您的模型已经过验证，则可以通过在“属性”窗口中将“对生成进行验证”属性设置为 false，在编译时取消验证。更改映射或模型时，可以暂时重新启用验证以验证更改。

请注意，对实体框架6的 Entity Framework Designer 进行了性能改进，“生成时验证”的成本要低于设计器的早期版本。

实体框架中的3种缓存

实体框架提供以下形式的缓存内置功能：

1. 对象缓存-在 `ObjectContext` 实例中内置的 `ObjectStateManager` 将跟踪使用该实例检索的对象的内存。这也称为第一级缓存。
2. 查询计划缓存-多次执行查询时重复使用生成的存储命令。
3. 元数据缓存-跨不同连接将模型的元数据共享到同一个模型。

除了 EF 提供的缓存外，还可以使用一种特殊类型的 ADO.NET 数据提供程序（称为包装提供程序），通过缓存为从数据库检索到的结果（也称为二级缓存）来扩展实体框架。

3.1 对象缓存

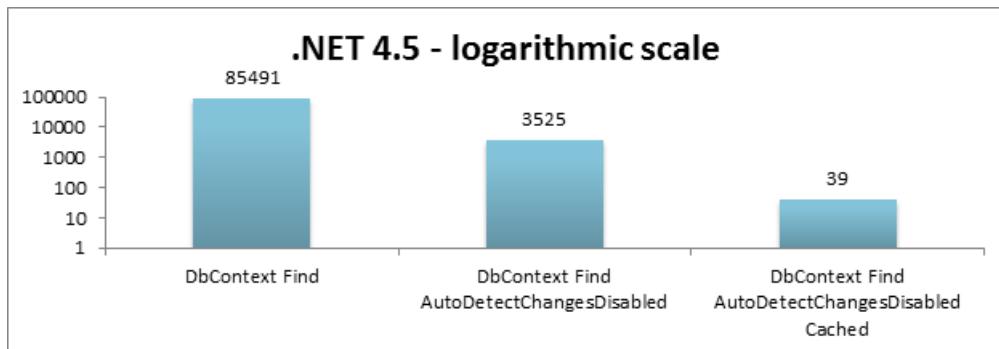
默认情况下，在查询结果中返回实体时，在 EF 具体化它之前，`ObjectContext` 会检查是否已将具有相同键的实体加载到其 `ObjectStateManager` 中。如果已存在具有相同键的实体，则 EF 会将其包含在查询结果中。尽管 EF 仍会对数据库发出查询，但这种行为可能会绕过多个时间具体化实体的开销。

使用 `DbContext` 查找从对象缓存获取实体的3.1.1

与常规查询不同，`DbSet` 中的 `Find` 方法（第一次在 EF 4.1 中包含的 API）将在内存中执行搜索，甚至对数据库发出查询。请注意，两个不同的 `ObjectContext` 实例将具有两个不同的 `ObjectStateManager` 实例，这意味着它们具有单独的对象缓存。

查找使用主键值来尝试查找上下文跟踪的实体。如果实体不在上下文中，则将针对数据库执行并评估查询，如果在上下文中或在数据库中找不到该实体，则返回 null。请注意，`Find` 还会返回已添加到上下文中但尚未保存到数据库中的实体。

使用“查找”时应考虑性能。默认情况下，对此方法的调用会触发对象缓存的验证，以检测仍处于挂起状态的对数据库的提交的更改。如果对象缓存或要添加到对象缓存中的大型对象图中有大量对象，则此过程可能会非常昂贵，但也可以禁用。在某些情况下，你可能会发现在禁用自动检测更改时调用 `Find` 方法时的差异数量级。但当对象实际在缓存中时，以及当必须从数据库中检索对象时，还可以看到另一种数量级。下面是一个示例图，其中使用了我们的一些 microbenchmarks（以毫秒为单位，以毫秒为单位）进行测量，负载为5000个实体：



禁用自动检测更改的查找示例：

```
context.Configuration.AutoDetectChangesEnabled = false;
var product = context.Products.Find(productId);
context.Configuration.AutoDetectChangesEnabled = true;
...
```

使用 Find 方法时必须考虑的事项如下：

- 如果对象不在缓存中，则会对 Find 的优点求反，但语法仍比按键查询更简单。
- 如果启用了自动检测更改，Find 方法的成本可能会增加一个数量级，甚至更多，具体取决于模型的复杂程度和对象缓存中的实体数量。

另外，请记住，Find 仅返回你要查找的实体，并且它不会自动加载其关联的实体（如果它们尚未在对象缓存中）。如果需要检索关联的实体，则可以通过键进行预先加载来使用查询。有关详细信息，请参阅[8.1 延迟加载与预先加载](#)。

当对象缓存具有多个实体时，[3.1.2 性能问题](#)

对象缓存有助于增加实体框架的总体响应能力。但是，当对象缓存加载的实体量非常大时，可能会影响某些操作，例如添加、删除、查找、条目、SaveChanges 等。特别是，触发对 DetectChanges 的调用的操作将受到超大型对象缓存的负面影响。DetectChanges 将对象关系图与对象状态管理器同步，其性能将由对象图的大小直接确定。有关 DetectChanges 的详细信息，请参阅[跟踪 POCO 实体中的更改](#)。

当使用实体框架6时，开发人员可以直接在 DbSet 上调用 AddRange 和 RemoveRange，而不是循环访问集合并为每个实例调用一次。使用范围方法的优点是，只为整个实体集支付一次 DetectChanges 成本，而不是每个添加的实体的费用。

3.2 查询计划缓存

第一次执行查询时，它会经历内部计划编译器，将概念性查询转换为存储命令（例如，在针对 SQL Server 运行时执行的 T-sql）。如果启用了查询计划缓存，则下一次执行查询时，将直接从查询计划缓存中检索 store 命令以便执行，绕过计划编译器。

查询计划缓存在同一 AppDomain 中跨 ObjectContext 实例共享。您无需保存到 ObjectContext 实例即可受益于查询计划缓存。

3.2.1 一些有关查询计划缓存的说明

- 查询计划缓存对于所有查询类型都是共享的：实体 SQL、LINQ to Entities 和 CompiledQuery 对象。
- 默认情况下，查询计划缓存针对实体 SQL 查询启用，无论是通过 EntityCommand 还是通过 ObjectQuery 执行。默认情况下，对于 .NET 4.5 实体框架上的 LINQ to Entities 查询，以及在实体框架6中，它也处于启用状态。
 - 可以通过将 EnablePlanCaching 属性（在 EntityCommand 或 ObjectQuery 上）设置为 false 来禁用查询计划缓存。例如：

```
var query = from customer in context.Customer
            where customer.CustomerId == id
            select new
            {
                customer.CustomerId,
                customer.Name
            };
ObjectQuery oQuery = query as ObjectQuery;
oQuery.EnablePlanCaching = false;
```

- 对于参数化查询，更改参数的值仍会命中缓存查询。但更改参数的方面（例如，大小、精度或小数位数）会命中缓存中的其他条目。
- 使用实体 SQL 时，查询字符串是密钥的一部分。更改查询会产生不同的缓存条目，即使查询在功能上是等效的。这包括更改大小写或空白。
- 使用 LINQ 时，将对查询进行处理以生成密钥的一部分。因此，更改 LINQ 表达式将生成不同的键。
- 其他技术限制可能适用；有关更多详细信息，请参阅 Autocompiled 查询。

3.2.2 缓存逐出算法

了解内部算法的工作方式将有助于您确定何时启用或禁用查询计划缓存。清理算法如下所示：

1. 缓存包含设置的条目数(800)后，我们会定期(每分钟一次)扫描缓存来启动一个计时器。
2. 在缓存扫描过程中，将从缓存中删除 LFRU (最不常使用的)条目。在确定要弹出的项时，此算法会考虑命中次数和使用期限。
3. 每次缓存扫描结束时，缓存将再次包含800个条目。

确定要逐出哪些项时，将同等对待所有缓存项。这意味着 CompiledQuery 的 store 命令与实体 SQL 查询的存储命令的逐出方式相同。

请注意，当缓存中存在800个实体时，将在中启动缓存逐出计时器，但缓存仅在此计时器启动后的60秒内进行扫描。这意味着缓存可能会增长到相当大的60秒。

3.2.3 测试指标演示查询计划缓存性能

为了说明查询计划缓存对应用程序性能的影响，我们执行了测试，在此测试中，我们对 Navision 模型执行了许多实体 SQL 查询。有关 Navision 模型的说明以及所执行的查询类型的说明，请参阅附录。在此测试中，我们首先遍历查询列表并执行每个查询，将它们添加到缓存(如果启用了缓存)。此步骤为 untimed。接下来，我们使主线程睡眠超过60秒，以允许进行缓存扫描;最后，我们会循环访问该列表，第二次执行缓存的查询。此外，在执行每组查询之前会刷新 SQL Server 计划缓存，以便我们准确地反映查询计划缓存提供的好处。

3.2.3.1 测试结果

II	EF5 III	EF5 II	EF6 III	EF6 II
枚举所有18723查询	124	125.4	124.3	125.3
避免扫描(仅限前800个查询，而不考虑复杂性)	41.7	5.5	40.5	5.4
仅限 AggregatingSubtotals 查询(total 178)	39.5	4.5	38.1	4.6

所有时间(以秒为单位)。

道德-执行大量不同的查询(例如动态创建的查询)时，缓存不起作用，并且生成的缓存刷新可使从计划缓存中获益最大的查询从实际使用到它。

AggregatingSubtotals 查询是我们测试过的查询中最复杂的查询。查询越复杂，查询计划缓存中的优势就越多。

由于 CompiledQuery 确实是缓存了其计划的 LINQ 查询，因此 CompiledQuery 与等效实体 SQL 查询的比较应具有类似的结果。事实上，如果应用具有大量的动态实体 SQL 查询，则使用查询填充缓存也会导致 CompiledQueries 从缓存中刷新时进行 "反编译"。在这种情况下，可以通过在动态查询上禁用缓存来设置 CompiledQueries 的优先级，从而提高性能。当然，更好的做法是重写应用程序以使用参数化查询，而不是动态查询。

3.3 使用 CompiledQuery 通过 LINQ 查询提高性能

我们的测试表明，使用 CompiledQuery 可以带来 7% over autocompiled LINQ 查询的好处;这意味着，从实体框架堆栈中执行代码的时间减少了 7%;这并不意味着应用程序的速度将更快7%。一般而言，与好处相比，在 EF 5.0 中编写和维护 CompiledQuery 对象的成本可能不值得。你的里程可能会有所不同，因此，如果你的项目需要额外的推送，则请执行此选项。请注意，CompiledQueries 仅与 ObjectContext 派生模型兼容，不与 DbContext 派生模型兼容。

有关创建和调用 CompiledQuery 的详细信息，请参阅[已编译的查询\(LINQ to Entities\)](#)。

在使用 CompiledQuery 时，需要考虑两个注意事项，即，要求使用静态实例和它们与可组合性相关的问题。下面对这两个注意事项进行了深入说明。

3.3.1 使用静态 CompiledQuery 实例

因为编译 LINQ 查询是一个耗时的过程，所以我们不希望每次需要从数据库中提取数据时都要这样做。使用 CompiledQuery 实例可以编译一次，并多次运行，但必须小心，并获得每次都重新使用同一个 CompiledQuery 实例，而不是反复编译它。需要使用静态成员存储 CompiledQuery 实例。否则，你将看不到任何权益。

例如，假设您的页面具有以下方法主体来处理显示所选类别的产品：

```
// Warning: this is the wrong way of using CompiledQuery
using (NorthwindEntities context = new NorthwindEntities())
{
    string selectedCategory = this.categoriesList.SelectedValue;

    var productsForCategory = CompiledQuery.Compile<NorthwindEntities, string, IQueryable<Product>>(
        (NorthwindEntities nwnd, string category) =>
        nwnd.Products.Where(p => p.Category.CategoryName == category)
    );

    this.productsGrid.DataSource = productsForCategory.Invoke(context, selectedCategory).ToList();
    this.productsGrid.DataBind();
}

this.productsGrid.Visible = true;
```

在这种情况下，每次调用该方法时，都将动态创建一个新的 CompiledQuery 实例。每次创建新的实例时，CompiledQuery 都将经历计划编译器，而不是通过从查询计划缓存中检索存储命令来查看性能优势。事实上，每次调用该方法时，都将使用新的 CompiledQuery 条目来污染查询计划缓存。

相反，您需要创建已编译查询的静态实例，以便在每次调用该方法时都要调用相同的编译查询。这样做的一种方法是将 CompiledQuery 实例添加为对象上下文的成员。然后，你可以通过 helper 方法访问 CompiledQuery 来使内容变得更干净：

```
public partial class NorthwindEntities : ObjectContext
{
    private static readonly Func<NorthwindEntities, string, IEnumerable<Product>> productsForCategoryCQ =
        CompiledQuery.Compile(
            (NorthwindEntities context, string categoryName) =>
            context.Products.Where(p => p.Category.CategoryName == categoryName)
        );

    public IEnumerable<Product> GetProductsForCategory(string categoryName)
    {
        return productsForCategoryCQ.Invoke(this, categoryName).ToList();
    }
}
```

此帮助器方法将按如下方式调用：

```
this.productsGrid.DataSource = context.GetProductsForCategory(selectedCategory);
```

3.3.2 在 CompiledQuery 上撰写

对任何 LINQ 查询进行组合的功能非常有用；为此，只需调用 IQueryable 后的方法，例如 *Skip ()* 或 *Count ()*。但这样做实际上会返回一个新的 IQueryable 对象。尽管从技术上讲，无需对 CompiledQuery 进行撰写，但这样做会导致生成新的 IQueryable 对象，该对象要求再次通过计划编译器。

某些组件将使用组合的 IQueryable 对象来启用高级功能。例如，ASP.NET 的 GridView 可以通过 SelectMethod 属性数据绑定到 IQueryable 对象。然后，GridView 将通过此 IQueryable 对象组合在一起，以允许对数据模型进行排序和分页。正如您所看到的，使用 GridView 的 CompiledQuery 不会命中已编译的查询，而是生成新的 autocompiled 查询。

在将渐进式筛选器添加到查询中时，您可能会遇到这种情况。例如，假设有一个“客户”页，其中包含用于可选筛选器（例如，Country 和 OrdersCount）的多个下拉列表。您可以在 CompiledQuery 的 IQueryable 结果中撰写这些筛选器，但这样做会导致新的查询每次执行时都会经历计划编译器。

```
using (NorthwindEntities context = new NorthwindEntities())
{
    IQueryable<Customer> myCustomers = context.InvokeCustomersForEmployee();

    if (this.orderCountFilterList.SelectedItem.Value != defaultFilterText)
    {
        int orderCount = int.Parse(orderCountFilterList.SelectedValue);
        myCustomers = myCustomers.Where(c => c.Orders.Count > orderCount);
    }

    if (this.countryFilterList.SelectedItem.Value != defaultFilterText)
    {
        myCustomers = myCustomers.Where(c => c.Address.Country == countryFilterList.SelectedValue);
    }

    this.customersGrid.DataSource = myCustomers;
    this.customersGrid.DataBind();
}
```

若要避免这种重新编译，可以重写 CompiledQuery 以考虑可能的筛选器：

```
private static readonly Func<NorthwindEntities, int, int?, string, IQueryable<Customer>>
customersForEmployeeWithFiltersCQ = CompiledQuery.Compile(
    (NorthwindEntities context, int empId, int? countFilter, string countryFilter) =>
    context.Customers.Where(c => c.Orders.Any(o => o.EmployeeID == empId))
    .Where(c => countFilter.HasValue == false || c.Orders.Count > countFilter)
    .Where(c => countryFilter == null || c.Address.Country == countryFilter)
);
```

这会在 UI 中调用，如：

```
using (NorthwindEntities context = new NorthwindEntities())
{
    int? countFilter = (this.orderCountFilterList.SelectedIndex == 0) ?
        (int?)null :
        int.Parse(this.orderCountFilterList.SelectedValue);

    string countryFilter = (this.countryFilterList.SelectedIndex == 0) ?
        null :
        this.countryFilterList.SelectedValue;

    IQueryable<Customer> myCustomers = context.InvokeCustomersForEmployeeWithFilters(
        countFilter, countryFilter);

    this.customersGrid.DataSource = myCustomers;
    this.customersGrid.DataBind();
}
```

这里的一个折衷是，生成的存储命令将始终包含带有 null 检查的筛选器，但对于数据库服务器而言，这些筛选器应该非常简单：

```
...
WHERE ((@0 = (CASE WHEN (@p_linq_1 IS NOT NULL) THEN cast(1 as bit) WHEN (@p_linq_1 IS NULL) THEN cast(0 as bit) END)) OR ([Project3].[C2] > @p_linq_2)) AND (@p_linq_3 IS NULL OR [Project3].[Country] = @p_linq_4)
```

3.4 元数据缓存

实体框架还支持元数据缓存。这实质上是在不同于同一模型的连接之间缓存类型信息和类型到数据库的映射信息。元数据缓存对于每个 AppDomain 都是唯一的。

3.4.1 元数据缓存算法

1. 模型的元数据信息存储在每个 EntityConnection 的 ItemCollection 中。
 - 为此, 模型的不同部分都有不同的 ItemCollection 对象。例如, StoreItemCollections 包含有关数据库模型的信息; ObjectItemCollection 包含有关数据模型的信息; EdmItemCollection 包含有关概念模型的信息。
2. 如果两个连接使用相同的连接字符串, 则它们将共享同一 ItemCollection 实例。
3. 在功能上等效但以字符为不同的连接字符串可能会导致不同的元数据缓存。我们确实标记了连接字符串, 因此只需更改令牌的顺序就会产生共享元数据。但在标记化后, 两个看起来相同的连接字符串可能不会计算为相同。
4. 定期检查 ItemCollection 的使用情况。如果确定工作区最近未访问过, 则会将其标记为在下次缓存扫描时清除。
5. 仅创建 EntityConnection 将导致创建元数据缓存(尽管在打开连接之前不会对其中的项集合进行初始化)。此工作区将保留在内存中, 直到缓存算法将其确定为 "正在使用"。

客户咨询团队已经编写了一篇博客文章, 其中介绍了如何保存对 ItemCollection 的引用, 以便在使用大型模型时避免 "弃用": <<http://blogs.msdn.com/b/appfabriccat/archive/2010/10/22/metadataworkspace-reference-in-wcf-services.aspx>>。

3.4.2 元数据缓存和查询计划缓存之间的关系

查询计划缓存实例驻留在 MetadataWorkspace 的存储区类型的 ItemCollection 中。这意味着缓存的存储命令将用于针对使用给定 MetadataWorkspace 实例化的任何上下文的查询。这也意味着, 如果有两个连接字符串略有不同, 并且在词汇切分后不匹配, 则会有不同的查询计划缓存实例。

3.5 结果缓存

使用结果缓存(也称为 "二级缓存"), 可以将查询结果保存在本地缓存中。发出查询时, 先查看在对应用商店进行查询之前, 结果是否可在本地使用。尽管实体框架不直接支持结果缓存, 但使用包装提供程序可以添加二级缓存。使用二级缓存的示例包装提供程序是 Alachisoft 的基于 NCache 实体框架二级缓存。

此第二级缓存的实现是在计算 LINQ 表达式后发生的插入功能(和 funcletized), 并且从第一级缓存计算或检索查询执行计划。然后, 第二级缓存将仅存储原始数据库结果, 因此具体化管道以后仍会执行。

3.5.1 用于包装提供程序的结果缓存的其他引用

- Julie Lerman 在实体框架和 Windows Azure 中编写了一个 "二级缓存", 其中包括如何更新示例包装提供程序以使用 Windows Server AppFabric 缓存: <https://msdn.microsoft.com/magazine/hh394143.aspx>
- 如果使用实体框架5, 则团队博客会提供一篇文章, 其中介绍了如何使用缓存提供程序为实体框架5: <<http://blogs.msdn.com/b/adonet/archive/2010/09/13/ef-caching-with-jarek-kowalski-s-provider.aspx>> 运行操作。它还包含 T4 模板, 以帮助自动将二级缓存添加到项目。

4 Autocompiled 查询

使用实体框架向数据库发出查询时, 必须完成一系列步骤, 然后才能真正具体化结果;一个这样的步骤就是查询编译。已知实体 SQL 查询具有良好的性能, 因为它们会自动缓存, 因此, 第二个或第三次执行相同的查询时, 可以跳过计划编译器并改用缓存的计划。

实体框架5也为 LINQ to Entities 查询引入了自动缓存。在过去的几个版本中实体框架创建 CompiledQuery 以提高性能, 因为这会使你的 LINQ to Entities 查询可缓存。因为缓存现在会自动执行, 而不使用 CompiledQuery, 因此, 我们会将此功能称为 "autocompiled 查询"。有关查询计划缓存及其机制的详细信息, 请参阅查询计划缓存。

实体框架检测需要重新编译查询的时间, 并在调用查询时进行调用(即使之前已编译过)。导致查询重新编译的常见情况如下:

- 更改与查询关联的 MergeOption。将不会使用缓存查询, 而是重新运行计划编译器, 并缓存新创建的计划。

- 更改 ContextOptions 的值。UseCSharpNullComparisonBehavior。与更改 MergeOption 的效果相同。

其他条件可能会阻止查询使用缓存。常见示例包括：

- 使用 `IEnumerable<T>.Contains(<T>)` (T 值)。
- 使用生成包含常量的查询的函数。
- 使用非映射对象的属性。
- 将查询链接到需要重新编译的另一个查询。

4.1 使用 `IEnumerable<T>.Contains(<T>)` (T 值)

实体框架不缓存调用 `IEnumerable<T>` 的查询。包含针对内存中集合的 `<T> (T 值)`，因为集合的值被视为可变的。下面的示例查询将不会被缓存，因此计划编译器将始终对其进行处理：

```
int[] ids = new int[10000];
...
using (var context = new MyContext())
{
    var query = context.MyEntities
        .Where(entity => ids.Contains(entity.Id));

    var results = query.ToList();
    ...
}
```

请注意，执行包含的 `IEnumerable` 的大小决定了查询的编译速度或速度。使用较大的集合（如上面的示例所示）时，性能可能会显著降低。

实体框架 6 包含对 `IEnumerable<T>` 方式的优化。包含 `<T> (T 值)` 在执行查询时工作。生成的 SQL 代码的速度要快得多，并且可读性更强，在大多数情况下，它在服务器上的执行速度更快。

4.2 使用生成包含常量的查询的函数

`Skip()`、`Take()`、`Contains()` 和 `DefaultIfEmpty()` LINQ 运算符不生成带参数的 SQL 查询，而是将作为常量传递给它们的值。因此，除非在后续的查询执行中使用了相同的常量，否则，最终可能会完全相同的查询将污染查询计划缓存（在 EF 堆栈和数据库服务器上）。例如：

```
var id = 10;
...
using (var context = new MyContext())
{
    var query = context.MyEntities.Select(entity => entity.Id).Contains(id);

    var results = query.ToList();
    ...
}
```

在此示例中，每次使用不同于 `id` 的值执行此查询时，查询都将编译为新计划。

特别要注意的是在进行分页时使用 `Skip` 和 `Take`。在 EF6 中，这些方法具有一个 lambda 重载，可有效地使缓存查询计划可重复使用，因为 EF 可以捕获传递给这些方法的变量并将其转换为 `SQLparameters`。这也有助于保留缓存清理器，因为另外，每个具有不同常量的查询都可用于 `Skip`，并会获取其自己的查询计划缓存条目。

请考虑以下代码，该代码是不理想的，但它只是为了求知欲此类查询：

```

var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i < count; ++i)
{
    var currentCustomer = customers.Skip(i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}

```

此相同代码的更快版本涉及到使用 lambda 调用 Skip:

```

var customers = context.Customers.OrderBy(c => c.LastName);
for (var i = 0; i < count; ++i)
{
    var currentCustomer = customers.Skip(() => i).FirstOrDefault();
    ProcessCustomer(currentCustomer);
}

```

由于每次运行查询时都使用相同的查询计划，因此第二个代码段的运行速度最多可达11%，这会节省 CPU 时间，并避免查询缓存的污染。此外，由于 Skip 参数在闭包中，代码可能如下所示：

```

var i = 0;
var skippyCustomers = context.Customers.OrderBy(c => c.LastName).Skip(() => i);
for (; i < count; ++i)
{
    var currentCustomer = skippyCustomers.FirstOrDefault();
    ProcessCustomer(currentCustomer);
}

```

4.3 使用非映射对象的属性

如果查询使用非映射对象类型的属性作为参数，则不会缓存查询。例如：

```

using (var context = new MyContext())
{
    var myObject = new NonMappedType();

    var query = from entity in context.MyEntities
                where entity.Name.StartsWith(myObject.MyProperty)
                select entity;

    var results = query.ToList();
    ...
}

```

在此示例中，假定类 NonMappedType 不是实体模型的一部分。可以轻松地将此查询更改为不使用非映射类型，而是使用局部变量作为查询的参数：

```

using (var context = new MyContext())
{
    var myObject = new NonMappedType();
    var myValue = myObject.MyProperty;
    var query = from entity in context.MyEntities
                where entity.Name.StartsWith(myValue)
                select entity;

    var results = query.ToList();
    ...
}

```

在这种情况下，查询将能够缓存，并将受益于查询计划缓存。

4.4 链接到需要重新编译的查询

按照上述示例所示，如果第二个查询依赖需要重新编译的查询，则还将重新编译整个第二个查询。下面是一个说明此方案的示例：

```
int[] ids = new int[10000];
...
using (var context = new MyContext())
{
    var firstQuery = from entity in context.MyEntities
                     where ids.Contains(entity.Id)
                     select entity;

    var secondQuery = from entity in context.MyEntities
                      where firstQuery.Any(otherEntity => otherEntity.Id == entity.Id)
                      select entity;

    var results = secondQuery.ToList();
    ...
}
```

示例是泛型的，但它说明了链接到 firstQuery 如何导致 secondQuery 无法缓存。如果 firstQuery 不是需要重新编译的查询，则 secondQuery 将被缓存。

5 NoTracking 查询

5.1 禁用更改跟踪以减少状态管理开销

如果你处于只读方案，并且想要避免将对象加载到 ObjectStateManager 中的开销，则可以发出 "无跟踪" 查询。可以在查询级别禁用更改跟踪。

但请注意，通过禁用更改跟踪，可以有效地关闭对象缓存。查询实体时，无法通过从 ObjectStateManager 中拉取先前具体化的查询结果来跳过具体化。如果在同一上下文中重复查询相同的实体，则可能会发现启用更改跟踪的性能优势。

使用 ObjectContext 进行查询时，ObjectQuery 和 ObjectSet 实例在设置后将记住 MergeOption，而在其上编写的查询将继承父查询的有效 MergeOption。当使用 DbContext 时，可以通过对 DbSet 调用 AsNoTracking() 修饰符来禁用跟踪。

5.1.1 在使用 DbContext 时禁用查询的更改跟踪

可以通过将对查询中的 AsNoTracking() 方法的调用链接到 NoTracking，将查询的模式切换为 NoTracking。与 ObjectQuery 不同，DbContext API 中的 DbSet 和 DbQuery 类没有 MergeOption 的可变属性。

```
var productsForCategory = from p in context.Products.AsNoTracking()
                           where p.Category.CategoryName == selectedCategory
                           select p;
```

5.1.2 使用 ObjectContext 禁用查询级别的更改跟踪

```
var productsForCategory = from p in context.Products
                           where p.Category.CategoryName == selectedCategory
                           select p;

((ObjectQuery)productsForCategory).MergeOption = MergeOption.NoTracking;
```

使用 ObjectContext 禁用整个实体集的更改跟踪 (5.1.3)

```

context.Products.MergeOption = MergeOption.NoTracking;

var productsForCategory = from p in context.Products
                           where p.Category.CategoryName == selectedCategory
                           select p;

```

5.2 测试度量值演示了 NoTracking 查询的性能优势

在此测试中，我们将通过比较跟踪到 Navision 模型的 NoTracking 查询来查看填充 ObjectStateManager 的成本。有关 Navision 模型的说明以及所执行的查询类型的说明，请参阅附录。在此测试中，我们循环访问一个查询列表并逐个执行。我们运行了两种不同的测试，一次是通过 NoTracking 查询，一次是 "AppendOnly" 的默认合并选项。每个变体运行3次，并采用运行的平均值。在测试之间，我们清除 SQL Server 上的查询缓存，并通过运行以下命令来缩减 tempdb：

1. DBCC DROPCLEANBUFFERS
2. DBCC FREEPROCCACHE
3. DBCC SHRINKDATABASE (tempdb, 0)

测试结果中，3个运行的中间值：

	Entity Framework 5	Entity Framework 6	ObjectStateManager	NoTracking
内存占用量	460361728	647127040	1163536 ms	596545536
运行时间	1273042 ms	195521 ms	832798720 ms	190228 ms

在运行结束时，实体框架5的内存占用量要低于实体框架6。实体框架6占用的额外内存是附加内存结构和可提高性能的代码的结果。

使用 ObjectStateManager 时，内存占用量也有明显差异。在跟踪从数据库具体化的所有实体时，实体框架5增加了30% 的占用量。执行此操作时，实体框架6增加了28%。

就时间而言，实体框架6在此测试中的实体框架5比大型边距高。实体框架6在实体框架5消耗的大约16% 的时间内完成了测试。此外，在使用 ObjectStateManager 时，实体框架5需要更多的时间来完成。相比之下，使用 ObjectStateManager 时，实体框架6使用了3% 的时间。

6 查询执行选项

实体框架提供了几种不同的查询方法。我们将查看以下选项，比较每个选项的优点和缺点，并检查其性能特征：

- LINQ to Entities。
- 无跟踪 LINQ to Entities。
- 通过 ObjectQuery 的实体 SQL。
- 通过 EntityCommand 的实体 SQL。
- System.data.objects.objectcontext.executestorequery.
- SqlQuery.
- CompiledQuery.

6.1 LINQ to Entities 查询

```

var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");

```

优点

- 适用于 CUD 操作。
- 完全具体化的对象。
- 最简单的方式是用编程语言内置的语法编写。
- 性能良好。

缺点

- 某些技术限制, 例如:
 - 对外部联接查询使用 System.Linq.Enumerable.DefaultIfEmpty 的模式将导致更复杂的查询, 而不是实体 SQL 中的简单外部联接语句。
 - 仍不能将 LIKE 与常规模式匹配一起使用。

6.2 无跟踪 LINQ to Entities 查询

当上下文派生 `ObjectContext` 时:

```
context.Products.MergeOption = MergeOption.NoTracking;  
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
```

当上下文派生 `DbContext`:

```
var q = context.Products.AsNoTracking()  
    .Where(p => p.Category.CategoryName == "Beverages");
```

优点

- 通过常规 LINQ 查询提高了性能。
- 完全具体化的对象。
- 最简单的方式是用编程语言内置的语法编写。

缺点

- 不适用于 CUD 操作。
- 某些技术限制, 例如:
 - 对外部联接查询使用 System.Linq.Enumerable.DefaultIfEmpty 的模式将导致更复杂的查询, 而不是实体 SQL 中的简单外部联接语句。
 - 仍不能将 LIKE 与常规模式匹配一起使用。

请注意, 即使未指定 `NoTracking`, 也不会跟踪项目标量属性的查询。例如:

```
var q = context.Products.Where(p => p.Category.CategoryName == "Beverages").Select(p => new { p.ProductName  
});
```

此特定查询未显式指定为 `NoTracking`, 但由于它不是对象状态管理器已知的类型, 因此不会跟踪具体化的结果。

6.3 实体 SQL ObjectQuery

```
ObjectQuery<Product> products = context.Products.Where("it.Category.CategoryName = 'Beverages'");
```

优点

- 适用于 CUD 操作。
- 完全具体化的对象。
- 支持查询计划缓存。

缺点

- 涉及与语言中内置的查询构造更容易出现用户错误的文本查询字符串。

6.4 通过 Entity 命令实体 SQL

```
EntityCommand cmd = eConn.CreateCommand();
cmd.CommandText = "Select p From NorthwindEntities.Products As p Where p.Category.CategoryName = 'Beverages'";

using (EntityDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
{
    while (reader.Read())
    {
        // manually 'materialize' the product
    }
}
```

优点

- 支持 .NET 4.0 中的查询计划缓存 (.NET 4.5 中的所有其他查询类型都支持计划缓存)。

缺点

- 涉及与语言中内置的查询构造更容易出现用户错误的文本查询字符串。
- 不适用于 CUD 操作。
- 结果不会自动具体化，必须从数据读取器读取。

6.5 SqlQuery 和 System.data.objects.objectcontext.executestorequery

数据库上的 SqlCommand :

```
// use this to obtain entities and not track them
var q1 = context.Database.SqlQuery<Product>("select * from products");
```

DbSet 上的 SqlCommand :

```
// use this to obtain entities and have them tracked
var q2 = context.Products.SqlQuery("select * from products");
```

ExecuteStoreQuery :

```
var beverages = context.ExecuteStoreQuery<Product>(
    @"      SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
    P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued, P.DiscontinuedDate
    FROM          Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE          (C.CategoryName = 'Beverages')"
);
```

优点

- 由于计划编译器被绕过，因此速度一般最快。
- 完全具体化的对象。
- 适用于从 DbSet 中使用的 CUD 操作。

缺点

- 查询的文本和容易出错。
- 查询通过使用存储语义而不是概念语义绑定到特定的后端。

- 存在继承时，手动查询需要考虑请求类型的映射条件。

6.6 CompiledQuery

```
private static readonly Func<NorthwindEntities, string, IQueryable<Product>> productsForCategoryCQ =
CompiledQuery.Compile(
    (NorthwindEntities context, string categoryName) =>
    context.Products.Where(p => p.Category.CategoryName == categoryName)
);

...
var q = context.InvokeProductsForCategoryCQ("Beverages");
```

优点

- 通过常规 LINQ 查询提高了7% 的性能改进。
- 完全具体化的对象。
- 适用于 CUD 操作。

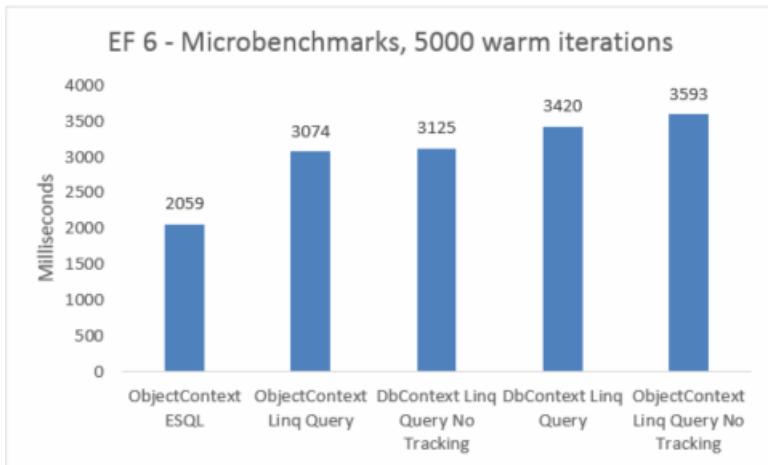
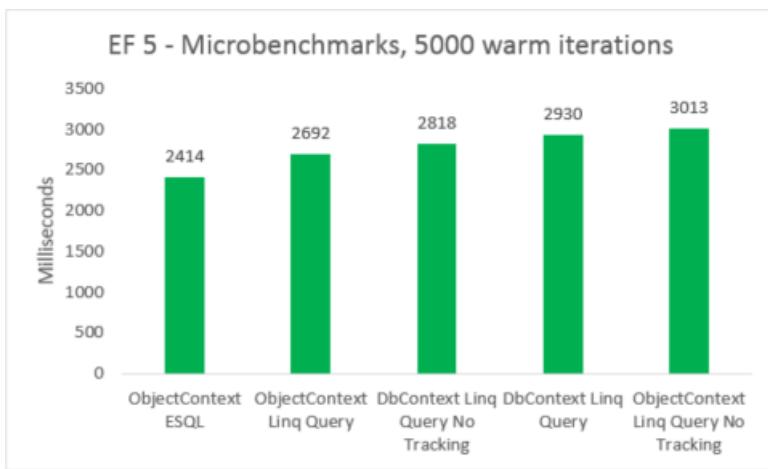
缺点

- 增加了复杂性和编程开销。
- 在已编译的查询顶部编写时，性能改进会丢失。
- 某些 LINQ 查询不能编写为 CompiledQuery (例如，匿名类型的投影)。

6.7 不同查询选项的性能比较

简单的 microbenchmarks，其中未计时上下文创建操作。对于受控环境中的一组非缓存实体，我们测量了5000次的查询。将会出现一条警告，其中包含警告：它们不反映应用程序产生的实际数量，而是对不同查询选项进行比较时的性能差异量的精确度量从苹果到苹果，不包括创建新上下文的成本。

EF	II	II(II)	II
EF5	ObjectContext ESQL	2414	38801408
EF5	ObjectContext Linq 查询	2692	38277120
EF5	DbContext Linq 查询无跟踪	2818	41840640
EF5	DbContext Linq 查询	2930	41771008
EF5	ObjectContext Linq 查询无跟踪	3013	38412288
EF6	ObjectContext ESQL	2059	46039040
EF6	ObjectContext Linq 查询	3074	45248512
EF6	DbContext Linq 查询无跟踪	3125	47575040
EF6	DbContext Linq 查询	3420	47652864
EF6	ObjectContext Linq 查询无跟踪	3593	45260800

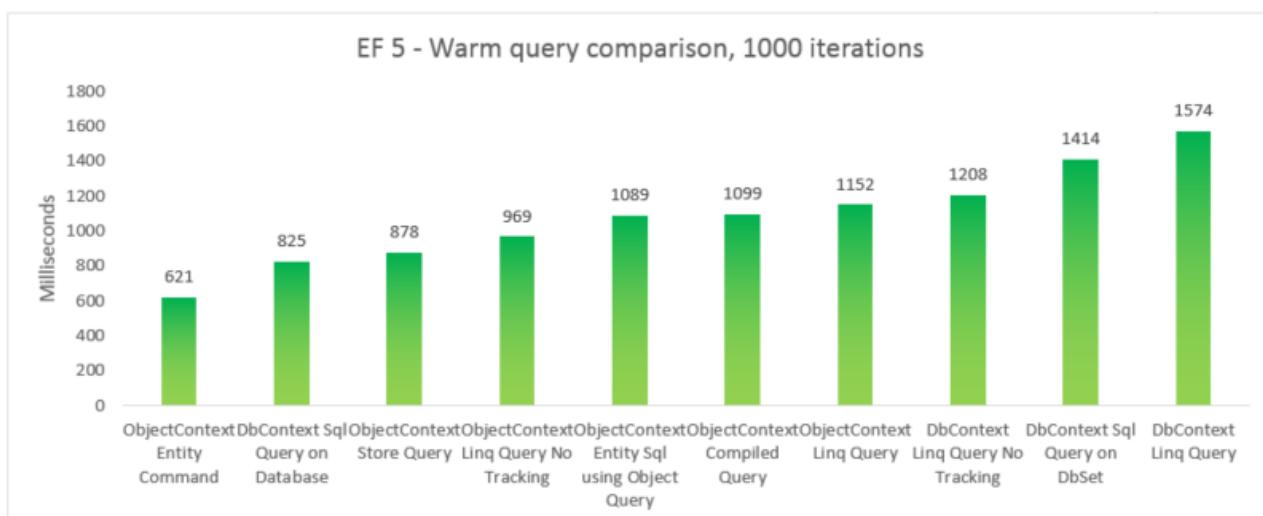


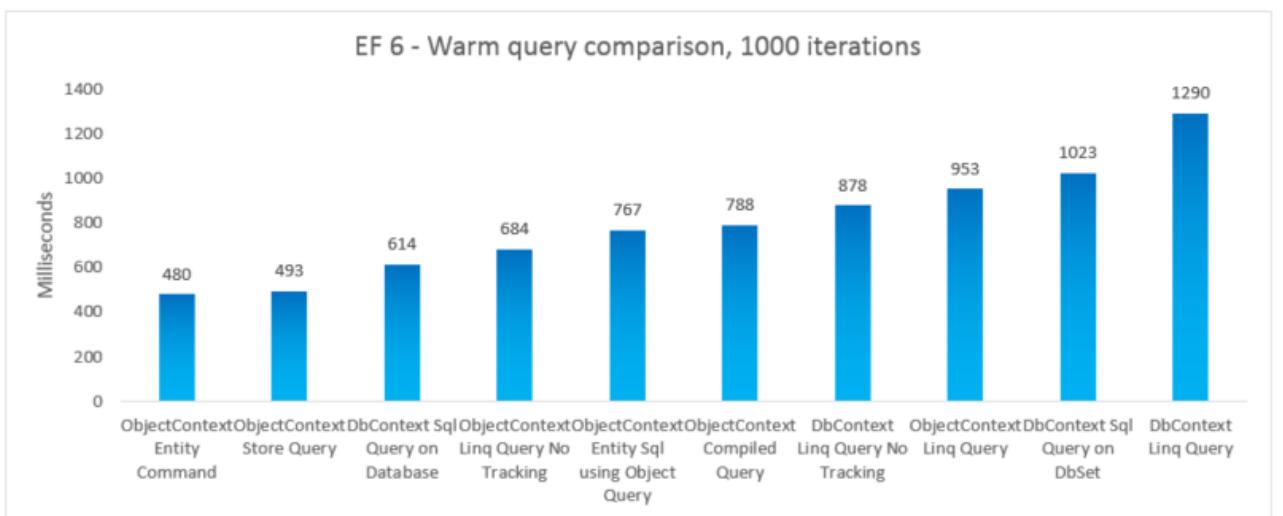
Microbenchmarks 对代码中的小更改非常敏感。在这种情况下，实体框架5与实体框架6的成本之间的差异是由于添加了[截取](#)和[事务改进](#)。不过，这些 microbenchmarks 数字是实体框架的作用的一小部分。在从实体框架5升级到实体框架6时，热查询的实际方案不应看到性能回归。

为了比较不同查询选项的实际性能，我们创建了5个单独的测试变体，其中使用不同的查询选项来选择类别名称为“饮料”的所有产品。每个迭代都包括创建上下文的成本，以及具体化所有返回实体的成本。在采用1000计时迭代的总和之前，将 untimed 运行10次迭代。显示的结果是从每个测试的5次运行中获取的中间值。有关详细信息，请参阅包含测试代码的附录 B。

EF	II	II(II)	II
EF5	ObjectContext 实体命令	621	39350272
EF5	对数据库进行 DbContext Sql 查询	825	37519360
EF5	ObjectContext 存储查询	878	39460864
EF5	ObjectContext Linq 查询无跟踪	969	38293504
EF5	使用对象查询的 ObjectContext 实体 Sql	1089	38981632
EF5	ObjectContext 编译查询	1099	38682624
EF5	ObjectContext Linq 查询	1152	38178816

EF		EF(EE)	
EF5	DbContext Linq 查询无跟踪	1208	41803776
EF5	DbSet 上的 DbContext Sql 查询	1414	37982208
EF5	DbContext Linq 查询	1574	41738240
EF6	ObjectContext 实体命令	480	47247360
EF6	ObjectContext 存储查询	493	46739456
EF6	对数据库进行 DbContext Sql 查询	614	41607168
EF6	ObjectContext Linq 查询无跟踪	684	46333952
EF6	使用对象查询的 ObjectContext 实体 Sql	767	48865280
EF6	ObjectContext 编译查询	788	48467968
EF6	DbContext Linq 查询无跟踪	878	47554560
EF6	ObjectContext Linq 查询	953	47632384
EF6	DbSet 上的 DbContext Sql 查询	1023	41992192
EF6	DbContext Linq 查询	1290	47529984





NOTE

为完整起见，我们提供了一个变体，用于在 EntityCommand 上执行实体 SQL 查询。但是，因为对于此类查询，结果不是具体化的，所以比较不一定是苹果。该测试包含一个接近近似值，旨在尝试进行比较 fairer。

在这种端到端的情况下，实体框架6比实体框架5高，因为在堆栈的几个部分中进行了性能改进，包括 DbContext 初始化更轻，<> 查找速度更快。

7设计时性能注意事项

7.1 继承策略

使用实体框架时的另一个性能注意事项是使用的继承策略。实体框架支持3种基本类型的继承及其组合：

- 每个层次结构一个表(TPH)–其中每个继承集都映射到一个具有鉴别器列的表，以指示该层次结构中的哪个特定类型在行中表示。
- 每种类型一个表(TPT)–其中每个类型在数据库中都有其自己的表;子表仅定义父表不包含的列。
- 每类表(TPC)–其中每个类型在数据库中都有其自己的完整表;子表定义其所有字段，包括在父类型中定义的字段。

如果模型使用 TPT 继承，则生成的查询将比使用其他继承策略生成的查询更复杂，这可能会导致存储上的执行时间较长。它通常需要更长的时间来生成针对 TPT 模型的查询并具体化生成的对象。

请参阅实体框架 "MSDN 博客文章：<<http://blogs.msdn.com/b/adonet/archive/2010/08/17/performance-considerations-when-using-tpt-table-per-type-inheritance-in-the-entity-framework.aspx>> 中" 使用 TPT 时的性能注意事项 "(每种类型的表)继承。

7.1.1 在 Model First 或 Code First 应用程序中避免了 TPT

在具有 TPT 架构的现有数据库上创建模型时，不会有太多选项。但在使用 Model First 或 Code First 创建应用程序时，应避免 TPT 继承，以解决性能问题。

使用 Entity Designer 向导中 Model First 时，将获取模型中的任何继承的 TPT。如果要使用 Model First 切换到 TPH 继承策略，则可以使用 Visual Studio 库中提供的 "Entity Designer 数据库生成 Power Pack" (<<http://visualstudiogallery.msdn.microsoft.com/df3541c3-d833-4b65-b942-989e7ec74c87/>>)。

当使用 Code First 配置模型与继承的映射时，默认情况下，EF 将使用 TPH，因此继承层次结构中的所有实体都将映射到同一个表。有关更多详细信息，请参阅 MSDN 杂志 (<http://msdn.microsoft.com/magazine/hh126815.aspx>) 中 "Code First 实体框架 4.1" 文章中的 "与熟知 API 的映射" 部分。

7.2 从 EF4 升级以改善模型生成时间

在实体框架5和6中提供了对生成模型的应用商店层(SSDL)的算法的特定于 SQL Server 的改进，并作为在安装

Visual Studio 2010 SP1 时实体框架4的更新。以下测试结果演示了生成非常大的模型时的改进，在本例中为 Navision 模型。有关详细信息，请参阅附录 C。

该模型包含1005个实体集和4227个关联集。

Visual Studio 版本	生成时间
Visual Studio 2010, 实体框架4	SSDL 生成:2小时27分钟 映射生成:1秒 CSDL 生成:1秒 ObjectLayer 生成:1秒 视图生成: 2 h 14 分钟
Visual Studio 2010 SP1, 实体框架4	SSDL 生成:1秒 映射生成:1秒 CSDL 生成:1秒 ObjectLayer 生成:1秒 视图生成:1小时53分钟
Visual Studio 2013, 实体框架5	SSDL 生成:1秒 映射生成:1秒 CSDL 生成:1秒 ObjectLayer 生成:1秒 视图生成:65分钟
Visual Studio 2013, 实体框架6	SSDL 生成:1秒 映射生成:1秒 CSDL 生成:1秒 ObjectLayer 生成:1秒 视图生成:28秒。

值得注意的是，在生成 SSDL 时，负载几乎完全花费在 SQL Server 中，而客户端开发计算机等待返回的结果是从服务器返回的。Dba 应该特别感谢这一改进。另外，值得注意的是，在生成模型的过程中，只需立即生成模型。

7.3 将大型模型与 Database First 和 Model First 分离

随着模型大小的增加，设计器图面变得混乱且难以使用。我们通常会考虑使用超过300个实体的模型太大，无法有效使用设计器。以下博客文章介绍了用于拆分大型模型的几个选项：

<<http://blogs.msdn.com/b/adonet/archive/2008/11/25/working-with-large-models-in-entity-framework-part-2.aspx>>。

张贴内容是为实体框架的第一个版本而编写的，但这些步骤仍适用。

7.4 实体数据源控件的性能注意事项

在多线程性能和压力测试中，我们已了解到，使用 EntityDataSource 控件的 web 应用程序的性能大大降低了这种情况。根本原因是，EntityDataSource 对 Web 应用程序引用的程序集重复调用 LoadFromAssembly，以发现要用作实体的类型。

解决方法是将 EntityDataSource 的 ContextTypeName 设置为派生 ObjectContext 类的类型名称。这会关闭用于扫描实体类型的所有引用程序集的机制。

设置 ContextTypeName 字段还可防止在 .NET 4.0 中的 EntityDataSource 无法通过反射从程序集加载类型时引发 ReflectionTypeLoadException 的功能问题。此问题已在 .NET 4.5 中解决。

7.5 POCO 实体和更改跟踪代理

实体框架使你能够将自定义数据类与数据模型一起使用，而无需对数据类本身进行任何修改。这意味着可以将“纯旧式”CLR 对象 (POCO)（例如，现有的域对象）与数据模型一起使用。这些 POCO 数据类（也称为持久性未知对象）映射到数据模型中定义的实体，支持与实体数据模型工具生成的实体类型相同的查询、插入、更新和删除行为。

实体框架还可以创建从 POCO 类型派生的代理类，当你想要在 POCO 实体上启用延迟加载和自动更改跟踪等功能时，可以使用这些类。POCO 类必须满足某些要求，才能允许实体框架使用代理，如下面所述：

<http://msdn.microsoft.com/library/dd468057.aspx>。

每次实体的属性值发生更改时，机会跟踪代理都会通知对象状态管理器，因此实体框架知道实体的实际状态。为此，可将通知事件添加到属性的 setter 方法体，并使对象状态管理器处理此类事件。请注意，创建代理实体通常比创建非代理 POCO 实体更昂贵，因为添加了实体框架创建的事件集。

如果 POCO 实体没有更改跟踪代理，则可以通过将实体的内容与以前保存的状态的副本进行比较来找到更改。当你在上下文中具有多个实体时，或者当你的实体具有非常多的属性（即使在上次进行比较之后它们都没有发生更改）时，此深层比较将成为一个漫长的过程。

摘要：创建更改跟踪代理时需支付性能，但当实体具有很多属性时，或在模型中有多个实体时，更改跟踪将有助于加速更改检测过程。对于包含少量属性的实体，这些实体的实体不会增长太多，具有更改跟踪代理的好处可能并不大。

8 加载相关实体

8.1 延迟加载与预先加载

实体框架提供多种不同的方法来加载与目标实体相关的实体。例如，在查询产品时，会有不同的方法将相关订单加载到对象状态管理器中。从性能角度来看，加载相关实体时要考虑的最重要问题是使用延迟加载还是预先加载。

使用预先加载时，会随目标实体集一起加载相关实体。在查询中使用 `Include` 语句来指示要引入的相关实体。

当使用延迟加载时，初始查询只会引入目标实体集。但只要您访问导航属性，就会针对该存储区发出另一个查询来加载相关实体。

加载实体后，该实体的任何进一步查询都将直接从对象状态管理器加载它，无论你使用的是延迟加载还是预先加载。

8.2 如何在延迟加载和预先加载之间进行选择

重要的是，您了解延迟加载与预先加载之间的差异，以便您可以为应用程序做出正确的选择。这将帮助您评估针对数据库的多个请求与可能包含大型有效负载的单个请求之间的权衡。在应用程序的某些部分使用预先加载和其他部分中的延迟加载可能是合适的。

例如，在此期间，假设您想要查询位于英国的客户及其订单计数。

使用预先加载

```
using (NorthwindEntities context = new NorthwindEntities())
{
    var ukCustomers = context.Customers.Include(c => c.Orders).Where(c => c.Address.Country == "UK");
    var chosenCustomer = AskUserToPickCustomer(ukCustomers);
    Console.WriteLine("Customer Id: {0} has {1} orders", chosenCustomer.CustomerID, chosenCustomer.Orders.Count);
}
```

使用延迟加载

```
using (NorthwindEntities context = new NorthwindEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;

    //Notice that the Include method call is missing in the query
    var ukCustomers = context.Customers.Where(c => c.Address.Country == "UK");

    var chosenCustomer = AskUserToPickCustomer(ukCustomers);
    Console.WriteLine("Customer Id: {0} has {1} orders", chosenCustomer.CustomerID, chosenCustomer.Orders.Count);
}
```

使用预先加载时，您将发出一个返回所有客户和所有订单的查询。存储命令如下所示：

```
SELECT
[Project1].[C1] AS [C1],
[Project1].[CustomerID] AS [CustomerID],
[Project1].[CompanyName] AS [CompanyName],
[Project1].[ContactName] AS [ContactName],
[Project1].[ContactTitle] AS [ContactTitle],
[Project1].[Address] AS [Address],
[Project1].[City] AS [City],
[Project1].[Region] AS [Region],
[Project1].[PostalCode] AS [PostalCode],
[Project1].[Country] AS [Country],
[Project1].[Phone] AS [Phone],
[Project1].[Fax] AS [Fax],
[Project1].[C2] AS [C2],
[Project1].[OrderID] AS [OrderID],
[Project1].[CustomerID1] AS [CustomerID1],
[Project1].[EmployeeID] AS [EmployeeID],
[Project1].[OrderDate] AS [OrderDate],
[Project1].[RequiredDate] AS [RequiredDate],
[Project1].[ShippedDate] AS [ShippedDate],
[Project1].[ShipVia] AS [ShipVia],
[Project1].[Freight] AS [Freight],
[Project1].[ShipName] AS [ShipName],
[Project1].[ShipAddress] AS [ShipAddress],
[Project1].[ShipCity] AS [ShipCity],
[Project1].[ShipRegion] AS [ShipRegion],
[Project1].[ShipPostalCode] AS [ShipPostalCode],
[Project1].[ShipCountry] AS [ShipCountry]
FROM (
    SELECT
        [Extent1].[CustomerID] AS [CustomerID],
        [Extent1].[CompanyName] AS [CompanyName],
        [Extent1].[ContactName] AS [ContactName],
        [Extent1].[ContactTitle] AS [ContactTitle],
        [Extent1].[Address] AS [Address],
        [Extent1].[City] AS [City],
        [Extent1].[Region] AS [Region],
        [Extent1].[PostalCode] AS [PostalCode],
        [Extent1].[Country] AS [Country],
        [Extent1].[Phone] AS [Phone],
        [Extent1].[Fax] AS [Fax],
        1 AS [C1],
        [Extent2].[OrderID] AS [OrderID],
        [Extent2].[CustomerID] AS [CustomerID1],
        [Extent2].[EmployeeID] AS [EmployeeID],
        [Extent2].[OrderDate] AS [OrderDate],
        [Extent2].[RequiredDate] AS [RequiredDate],
        [Extent2].[ShippedDate] AS [ShippedDate],
        [Extent2].[ShipVia] AS [ShipVia],
        [Extent2].[Freight] AS [Freight],
        [Extent2].[ShipName] AS [ShipName],
        [Extent2].[ShipAddress] AS [ShipAddress],
        [Extent2].[ShipCity] AS [ShipCity],
        [Extent2].[ShipRegion] AS [ShipRegion],
        [Extent2].[ShipPostalCode] AS [ShipPostalCode],
        [Extent2].[ShipCountry] AS [ShipCountry],
        CASE WHEN ([Extent2].[OrderID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C2]
    FROM [dbo].[Customers] AS [Extent1]
    LEFT OUTER JOIN [dbo].[Orders] AS [Extent2] ON [Extent1].[CustomerID] = [Extent2].[CustomerID]
    WHERE N'UK' = [Extent1].[Country]
) AS [Project1]
ORDER BY [Project1].[CustomerID] ASC, [Project1].[C2] ASC
```

使用延迟加载时，将最初发出以下查询：

```

SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
FROM [dbo].[Customers] AS [Extent1]
WHERE N'UK' = [Extent1].[Country]

```

每次访问客户的 "订单" 导航属性时，都会对该商店发出如下查询：

```

exec sp_executesql N'SELECT
[Extent1].[OrderID] AS [OrderID],
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[EmployeeID] AS [EmployeeID],
[Extent1].[OrderDate] AS [OrderDate],
[Extent1].[RequiredDate] AS [RequiredDate],
[Extent1].[ShippedDate] AS [ShippedDate],
[Extent1].[ShipVia] AS [ShipVia],
[Extent1].[Freight] AS [Freight],
[Extent1].[ShipName] AS [ShipName],
[Extent1].[ShipAddress] AS [ShipAddress],
[Extent1].[ShipCity] AS [ShipCity],
[Extent1].[ShipRegion] AS [ShipRegion],
[Extent1].[ShipPostalCode] AS [ShipPostalCode],
[Extent1].[ShipCountry] AS [ShipCountry]
FROM [dbo].[Orders] AS [Extent1]
WHERE [Extent1].[CustomerID] = @EntityKeyValue1',N'@EntityKeyValue1 nchar(5)',@EntityKeyValue1=N'AROUT'

```

有关详细信息，请参阅[加载相关对象](#)。

8.2.1 延迟加载与预先加载备忘单

这种情况并不完全适合选择预先加载和延迟加载。首先尝试了解两个策略之间的差异，以便您可以做出明智的决策；此外，如果你的代码符合以下任一方案，请考虑：

■	■
是否需要从提取的实体访问多个导航属性？	<p>■-这两个选项都可能会。但是，如果你的查询引入的负载并不太大，则使用预先加载可能会遇到性能优势，因为它需要较少的网络往返才能具体化你的对象。</p> <p>■-如果需要从实体访问多个导航属性，则可以通过在查询中使用多个 include 语句并预先加载来实现此目的。包含的实体越多，查询将返回的有效负载越大。将三个或更多实体添加到查询中后，请考虑切换到延迟加载。</p>
您是否确切知道运行时需要哪些数据？	<p>■-延迟加载将更适合你。否则，可能会最终查询不需要的数据。</p> <p>■-很可能是最好的选择；它有助于更快地加载整个集。如果查询需要提取大量数据，并且速度太慢，请改为尝试延迟加载。</p>

»

»

你的代码是否从数据库执行？(增加网络延迟)

■当网络延迟不是问题时，使用延迟加载可能会简化你的代码。请记住，应用程序的拓扑可能会发生更改，因此，不会对其进行已授予的数据邻近性。

■当网络出现问题时，只有你可以决定更适合你的方案。通常，预先加载将更好，因为它需要更少的往返行程。

多个包含8.2.2 的性能问题

当我们听到涉及服务器响应时间问题的性能问题时，问题的根源是经常用多个 Include 语句进行查询。虽然在查询中包括相关实体的功能非常强大，但必须了解所涉及的内容。

如果查询中包含多个 Include 语句，则需要相对较长的时间来生成存储命令。大多数情况下，尝试优化所生成的查询将花费大量时间。对于每个包含，生成的存储命令将包含外部联接或 Union，具体取决于你的映射。类似于这样的查询将在单个有效负载中引入数据库中的大型连接图形，这会 exacerbate 任何带宽问题，尤其是当有效负载中存在大量冗余时（例如，当使用多个级别的 Include 来遍历一对多方向的关联）。

可以通过使用 ToTraceString 访问查询的基础 TSQL，并在 SQL Server Management Studio 中执行存储命令来查看负载大小，来检查查询返回的负载是否过大的情况。在这种情况下，您可以尝试减少查询中包含语句的数量，以便只引入您需要的数据。您也可以将查询分解成较小的子查询序列，例如：

在中断查询之前：

```
using (NorthwindEntities context = new NorthwindEntities())
{
    var customers = from c in context.Customers.Include(c => c.Orders)
                    where c.LastName.StartsWith(lastNameParameter)
                    select c;

    foreach (Customer customer in customers)
    {
        ...
    }
}
```

中断查询后：

```
using (NorthwindEntities context = new NorthwindEntities())
{
    var orders = from o in context.Orders
                 where o.Customer.LastName.StartsWith(lastNameParameter)
                 select o;

    orders.Load();

    var customers = from c in context.Customers
                    where c.LastName.StartsWith(lastNameParameter)
                    select c;

    foreach (Customer customer in customers)
    {
        ...
    }
}
```

这仅适用于跟踪的查询，因为我们正在使用上下文自动执行标识解析和关联修正功能的能力。

与延迟加载一样，权衡更适用于较小的负载。你还可以使用各个属性的投影从每个实体显式选择所需的数据，但在这种情况下将不会加载实体，并且不支持更新。

用于获取属性延迟加载的8.2.3 解决方法

实体框架当前不支持标量或复杂属性的延迟加载。但是，如果您有一个包含大型对象(如 BLOB)的表，则可以使用表拆分将大属性分隔到一个单独的实体中。例如，假设您有一个包含 varbinary photo 列的产品表。如果不经常需要在查询中访问此属性，则可以使用表拆分来仅引入通常需要的实体部分。表示产品照片的实体仅在明确需要时才会加载。

显示如何启用表拆分的一个好资源是 Gil Fink 的 "实体框架中的表拆分" 博客文章：

<<http://blogs.microsoft.co.il/blogs/gilf/archive/2009/10/13/table-splitting-in-entity-framework.aspx>>。

9其他注意事项

9.1 服务器垃圾回收

当垃圾回收器未正确配置时，某些用户可能会遇到资源争用，这些争用会限制它们期望的并行度。每当在多线程方案中或在任何类似于服务器端系统的应用程序中使用 EF 时，请确保启用服务器垃圾回收。这是通过应用程序配置文件中的一个简单设置来完成的：

```
<?xmlversion="1.0" encoding="utf-8" ?>
<configuration>
    <runtime>
        <gcServer enabled="true" />
    </runtime>
</configuration>
```

这会减少线程争用，并增加吞吐量，使 CPU 饱和情况最多可达30%。通常情况下，应始终使用经典垃圾回收(更好地优化 UI 和客户端方案)以及服务器垃圾回收来测试应用程序的行为。

9.2 AutoDetectChanges

如前文所述，当对象缓存具有多个实体时，实体框架可能会显示性能问题。某些操作(例如添加、删除、查找、入口和 SaveChanges)触发对 DetectChanges 的调用，这些调用可能会根据对象缓存的大小而消耗大量 CPU。出现这种情况的原因是，对象缓存和对象状态管理器尝试在执行到上下文的每个操作上尽可能保持同步，以便在各种方案中保证生成的数据是正确的。

通常，在应用程序的整个生命周期中保留实体框架的自动更改检测是一种很好的做法。如果你的方案受到高 CPU 使用率的负面影响，并且你的配置文件指示原因是调用了 DetectChanges，请考虑在代码的敏感部分暂时关闭 AutoDetectChanges：

```
try
{
    context.Configuration.AutoDetectChangesEnabled = false;
    var product = context.Products.Find(productId);
    ...
}
finally
{
    context.Configuration.AutoDetectChangesEnabled = true;
}
```

在关闭 AutoDetectChanges 之前，最好了解这可能会导致实体框架无法跟踪有关在实体上发生的更改的特定信息。如果处理不当，这可能会导致应用程序的数据不一致。有关关闭 AutoDetectChanges 的详细信息，请参阅 <<http://blog.oneunicorn.com/2012/03/12/secrets-of-detectchanges-part-3-switching-off-automatic-detectchanges/>>。

9.3 每个请求的上下文

实体框架的上下文旨在用作生存期较短的实例，以提供最佳性能体验。上下文应为生存期较短且被丢弃，因此，在可能的情况下将其实现为非常轻量并 reutilize 元数据。在 web 方案中，请务必记住这一点，而不是将上下文用于单个请求的持续时间。同样，在非 web 应用场景中，应根据你对实体框架中不同缓存级别的了解来丢弃上下文。一般

而言，应避免在应用程序的整个生命周期中使用上下文实例，以及每个线程和静态上下文的上下文。

9.4 数据库 null 语义

默认情况下实体框架将生成具有 C# null 比较语义的 SQL 代码。请参考以下示例查询：

```
int? categoryId = 7;
int? supplierId = 8;
decimal? unitPrice = 0;
short? unitsInStock = 100;
short? unitsOnOrder = 20;
short? reorderLevel = null;

var q = from p in context.Products
        where p.Category.CategoryName == "Beverages"
            || (p.CategoryID == categoryId
                || p.SupplierID == supplierId
                || p.UnitPrice == unitPrice
                || p.UnitsInStock == unitsInStock
                || p.UnitsOnOrder == unitsOnOrder
                || p.ReorderLevel == reorderLevel)
        select p;

var r = q.ToList();
```

在此示例中，我们将多个可以为 null 的变量与实体上的可为 null 的属性进行比较，如供应商和单价。此查询生成的 SQL 将询问参数值与列值是否相同，或者参数和列值是否为 null。这将隐藏数据库服务器处理 null 值的方式，并将在不同的数据库供应商之间提供一致的 C# null 体验。另一方面，生成的代码有点复杂，如果查询的 where 语句中的比较量增长到了很大的数字，则可能无法正常运行。

处理这种情况的一种方法是使用数据库 null 语义。请注意，这可能会对 C# null 语义有不同的行为，因为现在实体框架将生成更简单的 SQL，它会公开数据库引擎处理空值的方式。对于上下文配置，可以对每个上下文激活数据库 null 语义，并提供一个配置行：

```
context.Configuration.UseDatabaseNullSemantics = true;
```

当使用数据库 null 语义时，小型到中等大小的查询不会显示明显的性能改进，但不同之处在于具有大量可能的 null 比较的查询会变得更加明显。

在上面的示例查询中，在受控环境中运行的 microbenchmark 中，性能差异低于 2%。

9.5 异步

实体框架 6 在 .NET 4.5 或更高版本上运行时，引入了对异步操作的支持。大多数情况下，具有 IO 相关争用的应用程序将从使用异步查询和保存操作中获益最多。如果你的应用程序不会受到 IO 争用的影响，则在最佳情况下，使用 `async` 会以同步方式运行，并以同步调用的相同时间量返回结果，或者在最坏情况下，只是将执行推迟到异步任务并添加额外的 time 到方案完成。

有关异步编程如何工作以帮助你确定 `async` 是否会提高应用程序的性能的详细信息，

请<http://msdn.microsoft.com/library/hh191443.aspx>。有关在实体框架上使用异步操作的详细信息，请参阅[Async Query And Save](#)。

9.6 NGEN

实体框架 6 不在 .NET Framework 的默认安装中。因此，默认情况下，实体框架程序集并不是 NGEN，这意味着所有实体框架代码都服从与任何其他 MSIL 程序集相同的 JIT'ing 成本。在生产环境中开发应用程序和冷启动应用程序时，这可能会降低 F5 体验。为了降低 JIT'ing 的 CPU 和内存成本，建议根据需要对实体框架映像进行 NGEN。有关如何改善实体框架 6 与 NGEN 的启动性能的详细信息，请参阅[通过 Ngen 改善启动性能](#)。

9.7 Code First 与 EDMX

通过内存中的概念模型(对象)、存储架构(数据库)和映射之间的映射，实体框架面向对象的编程与关系数据库之间

的阻抗不匹配问题的原因。双向。对于 short，此元数据称为实体数据模型或 EDM。在此 EDM 中，实体框架将派生视图，以将数据从内存中的对象往返到数据库并返回数据。

如果将实体框架用于正式指定了概念模型、存储架构和映射的 EDMX 文件，则模型加载阶段仅需验证 EDM 是否正确（例如，确保没有缺少映射），然后生成视图，然后验证视图，并使此元数据可供使用。只有这样才能执行查询，或者将新数据保存到数据存储区。

Code First 方法就是一种复杂的实体数据模型生成器。实体框架必须从提供的代码生成 EDM；它通过分析模型中涉及的类、应用约定并通过流畅的 API 配置模型来实现此目的。生成 EDM 后，实体框架本质上的行为与在项目中存在 EDMX 文件的方式相同。因此，从 Code First 生成模型会增加额外的复杂性，与具有 EDMX 相比，实体框架的启动时间更慢。成本完全取决于正在生成的模型的大小和复杂程度。

选择使用 EDMX 与 Code First 时，必须知道 Code First 引入的灵活性会增加第一次生成模型时的成本。如果你的应用程序可以承受此首次加载的成本，则通常 Code First 将是首选方法。

10 调查性能

10.1 使用 Visual Studio 探查器

如果实体框架存在性能问题，则可以使用类似于 Visual Studio 中内置的探查器来查看应用程序所花费的时间。这是一种工具，用于在“浏览 ADO.NET 实体框架-第1部分”博客文章

(<http://blogs.msdn.com/b/adonet/archive/2008/02/04/exploring-the-performance-of-the-ado-net-entity-framework-part-1.aspx>)，其中显示实体框架在冷和温查询过程中花费的时间。

“使用 Visual Studio 2010 探查器分析实体框架”博客文章由数据和建模客户咨询团队介绍了如何使用探查器来调查性能问题的实际示例。<http://blogs.msdn.com/b/dmcat/archive/2010/04/30/profiling-entity-framework-using-the-visual-studio-2010-profiler.aspx>。此帖子是为 windows 应用程序编写的。如果需要对 web 应用程序进行分析，Windows 性能记录器(“)和 Windows 性能分析器(WPA)工具的工作方式可能比 Visual Studio 中的更好。“和 WPA 是 windows 评估和部署工具包(<http://www.microsoft.com/download/details.aspx?id=39982>)随附的 Windows 性能工具包的一部分。

10.2 应用程序/数据库分析

Visual Studio 中内置的探查器会告诉你应用程序花费时间的位置。可以使用另一种类型的探查器，根据需要对正在生产或预生产环境中的运行中的应用程序执行动态分析，并查找数据库访问的常见缺陷和反模式。

两个商用探查器是实体框架探查器(<http://efprof.com>) 和 ORMPProfiler (<http://ormprofiler.com>)。

如果你的应用程序是使用 Code First 的 MVC 应用程序，则可以使用 Stackexchange.redis 的 MiniProfiler。Scott Hanselman 在他的博客中介绍了此工具：

<http://www.hanselman.com/blog/NuGetPackageOfTheWeek9ASPNETMiniProfilerFromStackExchangeRocksYourWorld.aspx>。

有关分析应用程序的数据库活动的详细信息，请参阅 Julie Lerman 的 MSDN 杂志文章，其中标题[实体框架中的“分析数据库”活动](#)。

10.3 数据库记录器

如果使用实体框架 6，还应考虑使用内置日志记录功能。可以通过简单的单行配置指示上下文的数据库属性记录其活动：

```
using (var context = newQueryComparison.DbC.NorthwindEntities())
{
    context.Database.Log = Console.WriteLine;
    var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
    q.ToList();
}
```

在此示例中，数据库活动将记录到控制台中，但 Log 属性可以配置为调用<字符串>委托的任何操作。

如果要在不重新编译的情况下启用数据库日志记录，并且使用实体框架6.1或更高版本，则可以通过在应用程序的 web.config 或 app.config 文件中添加侦听器来执行此操作。

```
<interceptors>
  <interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">
    <parameters>
      <parameter value="C:\Path\To\My\LogOutput.txt"/>
    </parameters>
  </interceptor>
</interceptors>
```

有关如何添加不进行重新编译的日志记录的详细信息，请参阅 <http://blog.oneunicorn.com/2014/02/09/ef-6-1-turning-on-logging-without-recompiling/>。

11附录

11.1 A. 测试环境

此环境将两台计算机的安装程序与客户端应用程序单独的计算机上的数据库一起使用。计算机位于同一机架中，因此网络延迟相对较低，但比单计算机环境更为现实。

11.1.1 应用服务器

11.1.1.1 软件环境

- 实体框架4软件环境
 - OS 名称: Windows Server 2008 R2 Enterprise SP1。
 - Visual Studio 2010 –旗舰版。
 - Visual Studio 2010 SP1 (仅适用于某些比较)。
- 实体框架5和6软件环境
 - OS 名称: Windows 8.1 Enterprise
 - Visual Studio 2013 –旗舰版。

11.1.1.2 硬件环境

- 双处理器: Intel (R) L5520 W3530 @ 2.27 GHz, 2261 Mhz8 GHz, 4核, 84逻辑处理器。
- 2412 GB RamRAM。
- 136 GB SCSI250GB SATA 7200 rpm 3GB/s 驱动器拆分为4个分区。

11.1.2 DB 服务器

11.1.2.1 软件环境

- OS 名称: Windows Server 2008 R 28.1 Enterprise SP1。
- SQL Server 2008 R22012.

11.1.2.2 硬件环境

- 单处理器: Intel (R) 至强(R) CPU L5520 @ 2.27 GHz, 2261 MhzES-1620 0 @ 3.60 GHz, 4核, 8个逻辑处理器。
- 824 GB RamRAM。
- 465 GB ATA500GB SATA 7200 rpm 6GB/s 驱动器拆分为4个分区。

11.2 b. 查询性能比较测试

Northwind 模型用于执行这些测试。它是使用实体框架设计器从数据库生成的。然后，使用以下代码来比较查询执行选项的性能：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Common;
using System.Data.Entity.Infrastructure;
using System.Data.EntityClient;
using System.Data.Objects;
using System.Linq;
```

```

namespace QueryComparison
{
    public partial class NorthwindEntities : ObjectContext
    {
        private static readonly Func<NorthwindEntities, string, IQueryable<Product>> productsForCategoryCQ =
CompiledQuery.Compile(
            (NorthwindEntities context, string categoryName) =>
            context.Products.Where(p => p.Category.CategoryName == categoryName)
        );

        public IQueryable<Product> InvokeProductsForCategoryCQ(string categoryName)
        {
            return productsForCategoryCQ(this, categoryName);
        }
    }

    public class QueryTypePerfComparison
    {
        private static string entityConnectionString =
@"metadata=res://*/Northwind.csdl|res://*/Northwind.ssdl|res://*/Northwind.msl;provider=System.Data.SqlClient;
provider connection string='data source=.;initial catalog=Northwind;integrated
security=True;multipleactiveresultsets=True;App=EntityFramework'";

        public void LINQIncludingContextCreation()
        {
            using (NorthwindEntities context = new NorthwindEntities())
            {
                var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
                q.ToList();
            }
        }

        public void LINQNoTracking()
        {
            using (NorthwindEntities context = new NorthwindEntities())
            {
                context.Products.MergeOption = MergeOption.NoTracking;

                var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
                q.ToList();
            }
        }

        public void CompiledQuery()
        {
            using (NorthwindEntities context = new NorthwindEntities())
            {
                var q = context.InvokeProductsForCategoryCQ("Beverages");
                q.ToList();
            }
        }

        public void ObjectQuery()
        {
            using (NorthwindEntities context = new NorthwindEntities())
            {
                ObjectQuery<Product> products = context.Products.Where("it.Category.CategoryName =
'Beverages'");
                products.ToList();
            }
        }

        public void EntityCommand()
        {
            using (EntityConnection eConn = new EntityConnection(entityConnectionString))
            {
                eConn.Open();
                EntityCommand cmd = eConn.CreateCommand();

```

```

        cmd.CommandText = "Select p From NorthwindEntities.Products As p Where p.Category.CategoryName
= 'Beverages'";

        using (EntityDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
        {
            List<Product> productsList = new List<Product>();
            while (reader.Read())
            {
                DbDataRecord record = (DbDataRecord)reader.GetValue(0);

                // 'materialize' the product by accessing each field and value. Because we are
materializing products, we won't have any nested data readers or records.
                int fieldCount = record.FieldCount;

                // Treat all products as Product, even if they are the subtype DiscontinuedProduct.
                Product product = new Product();

                product.ProductID = record.GetInt32(0);
                product.ProductName = record.GetString(1);
                product.SupplierID = record.GetInt32(2);
                product.CategoryID = record.GetInt32(3);
                product.QuantityPerUnit = record.GetString(4);
                product.UnitPrice = record.GetDecimal(5);
                product.UnitsInStock = record.GetInt16(6);
                product.UnitsOnOrder = record.GetInt16(7);
                product.ReorderLevel = record.GetInt16(8);
                product.Discontinued = record.GetBoolean(9);

                productsList.Add(product);
            }
        }
    }

    public void ExecuteStoreQuery()
    {
        using (NorthwindEntities context = new NorthwindEntities())
        {
            ObjectResult<Product> beverages = context.ExecuteStoreQuery<Product>(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
    FROM        Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE       (C.CategoryName = 'Beverages')"
);
            beverages.ToList();
        }
    }

    public void ExecuteStoreQueryDbContext()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var beverages = context.Database.SqlQuery<QueryComparison.DbC.Product>(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
    FROM        Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE       (C.CategoryName = 'Beverages')"
);
            beverages.ToList();
        }
    }

    public void ExecuteStoreQueryDbSet()
    {
        using (var context = new QueryComparison.DbC.NorthwindEntities())
        {
            var beverages = context.Products.SqlQuery(
@"
    SELECT      P.ProductID, P.ProductName, P.SupplierID, P.CategoryID, P.QuantityPerUnit, P.UnitPrice,
P.UnitsInStock, P.UnitsOnOrder, P.ReorderLevel, P.Discontinued
"
);
        }
    }
}

```

```

    FROM          Products AS P INNER JOIN Categories AS C ON P.CategoryID = C.CategoryID
    WHERE         (C.CategoryName = 'Beverages')"
);

        beverages.ToList();
    }

}

public void LINQIncludingContextCreationDbContext()
{
    using (var context = new QueryComparison.DbC.NorthwindEntities())
    {
        var q = context.Products.Where(p => p.Category.CategoryName == "Beverages");
        q.ToList();
    }
}

public void LINQNoTrackingDbContext()
{
    using (var context = new QueryComparison.DbC.NorthwindEntities())
    {
        var q = context.Products.AsNoTracking().Where(p => p.Category.CategoryName == "Beverages");
        q.ToList();
    }
}
}

```

11.3 Navision 模型

Navision 数据库是用于演示 Microsoft Dynamics – 导航的大型数据库。生成的概念模型包含 1005 个实体集和 4227 个关联集。测试中使用的模型为 "平面" – 未向其添加继承。

用于 Navision 测试的 11.3.1 查询

与 Navision 模型一起使用的查询列表包含 3 类实体 SQL 查询：

11.3.1.1 查找

不带聚合的简单查找查询

- 计数：16232
- 示例：

```

<Query complexity="Lookup">
    <CommandText>Select value distinct top(4) e.Idle_Time From NavisionFKContext.Session as e</CommandText>
</Query>

```

11.3.1.2 SingleAggregating

具有多个聚合但没有小计（单个查询）的普通 BI 查询

- 计数：2313
- 示例：

```

<Query complexity="SingleAggregating">
    <CommandText>NavisionFK.MDF_SessionLogin_Time_Max()</CommandText>
</Query>

```

其中 MDF_SessionLogin_Time_Max() 在模型中定义为：

```

<Function Name="MDF_SessionLogin_Time_Max" ReturnType="Collection(DateTime)">
  <DefiningExpression>SELECT VALUE Edm.Min(E.Login_Time) FROM NavisionFKContext.Session as E
</DefiningExpression>
</Function>

```

11.3.1.3 AggregatingSubtotals

具有聚合和小计的 BI 查询(通过 union all)

- 计数:178

- 示例:

```

<Query complexity="AggregatingSubtotals">
  <CommandText>
using NavisionFK;
function AmountConsumed(entities Collection([CRONUS_International_Ltd__Zone])) as
(
  Edm.Sum(select value N.Block_Movement FROM entities as E, E.CRONUS_International_Ltd__Bin as N)
)
function AmountConsumed(P1 Edm.Int32) as
(
  AmountConsumed(select value e from NavisionFKContext.CRONUS_International_Ltd__Zone as e where
e.Zone_Ranking = P1)
)

-----
(
  select top(10) Zone_Ranking, Cross_Dock_Bin_Zone, AmountConsumed(GroupPartition(E))
  from NavisionFKContext.CRONUS_International_Ltd__Zone as E
  where AmountConsumed(E.Zone_Ranking) > @MinAmountConsumed
  group by E.Zone_Ranking, E.Cross_Dock_Bin_Zone
)
union all
(
  select top(10) Zone_Ranking, Cast(null as Edm.Byte) as P2, AmountConsumed(GroupPartition(E))
  from NavisionFKContext.CRONUS_International_Ltd__Zone as E
  where AmountConsumed(E.Zone_Ranking) > @MinAmountConsumed
  group by E.Zone_Ranking
)
union all
{
  Row(Cast(null as Edm.Int32) as P1, Cast(null as Edm.Byte) as P2, AmountConsumed(select value E
from
NavisionFKContext.CRONUS_International_Ltd__Zone as E
where AmountConsumed(E.Zone_Ranking)
> @MinAmountConsumed))
}</CommandText>
<Parameters>
  <Parameter Name="MinAmountConsumed" DbType="Int32" Value="10000" />
</Parameters>
</Query>

```

通过 NGen 改善启动性能

2020/3/11 ·

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

.NET Framework 支持为托管应用程序和库生成本机映像，以帮助应用程序的启动速度更快，在某些情况下还会使用更少的内存。本机映像是通过在执行应用程序之前将托管代码程序集转换为包含本机计算机指令的文件创建的，从而使 .NET JIT (实时) 编译器不必在应用程序运行时。

在版本 6 之前，EF 运行时的核心库是 .NET Framework 的一部分，并且会为其自动生成本机映像。从版本 6 开始，整个 EF 运行时已合并到 EntityFramework NuGet 包中。本机映像现在必须使用 Ngen.exe 命令行工具生成，才能获得类似的结果。

经验观察显示 EF 运行时程序集的本机映像可以在应用程序启动时间的 1 到 3 秒之间缩短。

如何使用 Ngen.exe

Ngen.exe 工具的最基本功能是为程序集及其所有直接依赖项“安装”（即创建和保留到磁盘）本机映像。下面介绍如何实现此目的：

1. 以管理员身份打开“命令提示符”窗口。
2. 将当前工作目录更改为你要为其生成本机映像的程序集的位置：

```
cd <*Assemblies location>
```

3. 根据你的操作系统和应用程序的配置，可能需要为 32 位体系结构、64 位体系结构或两者生成本机映像。

对于 32 位运行：

```
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\ngen install <Assembly name>
```

对于 64 位运行：

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\ngen install <Assembly name>
```

TIP

生成错误体系结构的本机映像是一个非常常见的错误。如果有疑问，只需为适用于计算机中安装的操作系统的所有体系结构生成本机映像。

Ngen.exe 还支持其他功能，例如卸载和显示安装的本机映像、对多个映像的生成进行排队等。有关用法的详细信息，请参阅[ngen.exe 文档](#)。

何时使用 Ngen.exe

当决定在基于 EF 版本 6 或更高版本的应用程序中为哪些程序集生成本机映像时，应考虑以下选项：

- **主要 ef 运行时程序集 EntityFramework**: 基于 EF 的典型应用程序在启动时或在首次访问数据库时, 将从该程序集执行大量代码。因此, 创建此程序集的本机映像将产生最大的启动性能提升。
- **应用程序使用的任何 EF 提供程序程序集**: 启动时间也可能与生成这些的本机映像稍微有利。例如, 如果应用程序使用 EF 提供程序进行 SQL Server 则需要为 EntityFramework 生成本机映像。
- **应用程序的程序集和其他依赖项**: [ngen.exe 文档](#)介绍了用于选择要为其生成本机映像的程序集以及本机映像对安全性的影响、高级选项(如在调试和分析方案中使用本机映像等)的一般条件。

TIP

请确保仔细衡量对应用程序的启动性能和总体性能的使用本机映像的影响, 并将其与实际要求进行比较。尽管本机映像通常有助于改善启动性能, 但在某些情况下, 会降低内存使用率, 并非所有方案都将同样受益。例如, 在稳定状态执行(即, 应用程序使用的所有方法都被调用了至少一次), JIT 编译器生成的代码实际上会产生比本机映像更好的性能。

在开发计算机中使用 Ngen.exe

在开发期间, .NET JIT 编译器将为经常更改的代码提供最佳总体平衡。对于已编译的依赖项(如 EF 运行时程序集), 生成本机映像可通过在每次执行开始时缩短几秒钟来帮助加速开发和测试。

查找 EF 运行时程序集的好地方是解决方案的 NuGet 包位置。例如, 对于使用 EF 6.0.2 与 SQL Server 并面向 .NET 4.5 或更高版本的应用程序, 你可以在命令提示符窗口中键入以下内容(请记得以管理员身份打开它):

```
cd <Solution directory>\packages\EntityFramework.6.0.2\lib\net45
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\ngen install EntityFramework.SqlServer.dll
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\ngen install EntityFramework.SqlServer.dll
```

NOTE

这就充分利用了为 EF 提供程序安装本机映像 SQL Server 的事实, 在默认情况下, 还会安装主要 EF 运行时程序集的本机映像。这是因为 Ngen.exe 可以检测到 EntityFramework 是位于同一个目录中 EntityFramework 程序集的直接依赖项。

在安装过程中创建本机映像

在安装过程中, WiX 工具包支持为托管程序集生成本机映像的排队, 如本操作[方法指南](#)中所述。另一种方法是创建执行 Ngen.exe 命令的自定义安装任务。

正在验证是否正在将本机映像用于 EF

您可以通过查找扩展名为 ".ni.dll" 或 ".al.exe" 的加载程序集来验证特定应用程序是否正在使用本机程序集。例如, EF 的主运行时程序集的本机映像将称为 EntityFramework。检查进程的加载的 .NET 程序集的一种简单方法是使用[进程资源管理器](#)。

其他需要注意的问题

创建程序集的本机映像不应与在 GAC 中注册程序集([全局程序集缓存](#))混淆。Ngen.exe 允许创建不在 GAC 中的程序集的映像, 事实上, 使用特定 EF 版本的多个应用程序可以共享同一个本机映像。虽然 Windows 8 可以自动为 GAC 中的程序集创建本机映像, 但不建议将 EF 运行时与应用程序一起部署, 并且我们不建议在 GAC 中注册它, 因为这样会对程序集解析产生负面影响, 在其他方面维护应用程序。

预生成的映射视图

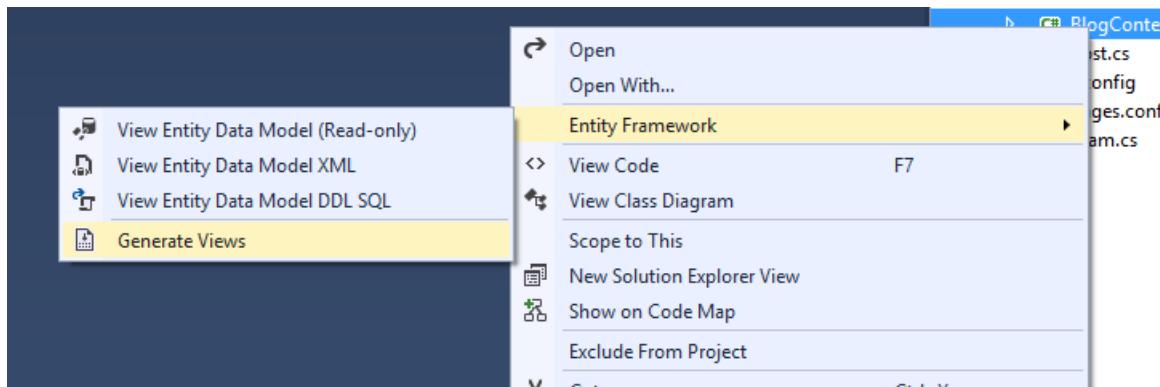
2020/3/11 •

在实体框架可以执行查询或将更改保存到数据源之前，它必须生成一组用于访问数据库的映射视图。这些映射视图是一组实体 SQL 语句，它们以抽象的方式表示数据库，是每个应用程序域缓存的元数据的一部分。如果在同一个应用程序域中创建同一个上下文的多个实例，则它们将重用缓存的元数据中的映射视图，而不是重新生成它们。由于映射视图生成是执行第一个查询的总体成本的重要部分，因此实体框架允许您预先生成映射视图，并将它们包含在已编译的项目中。有关详细信息，请参阅 [性能注意事项\(实体框架\)](#)。

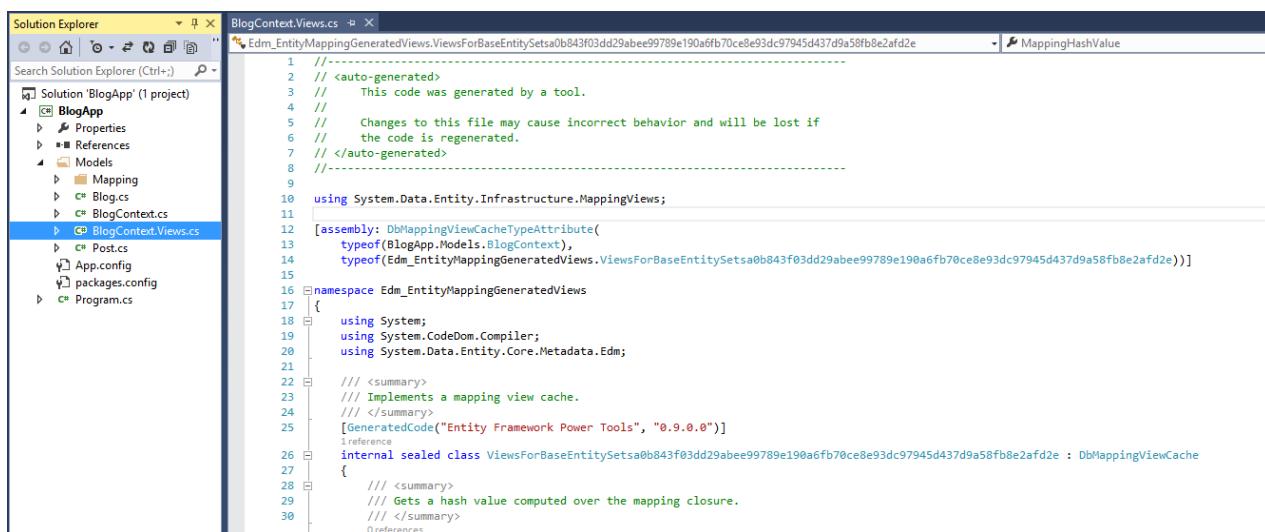
用 EF Power Tools 社区版生成映射视图

预生成视图的最简单方法是使用[EF Power Tools 社区版](#)。安装了 Power Tools 后，将有一个用于生成视图的菜单选项，如下所示。

- 对于Code First模型，右键单击包含 DbContext 类的代码文件。
- 对于EF 设计器模型，右键单击 EDMX 文件。



完成此过程后，您将拥有一个类似于下面生成的类



现在，当你运行应用程序 EF 时，将使用此类根据需要加载视图。如果模型发生更改，并且不重新生成此类，则 EF 会引发异常。

从代码生成映射视图-EF6 向前

生成视图的另一种方法是使用 EF 提供的 API。使用此方法时，您可以自由地序列化视图，但您也需要自行加载视图。

NOTE

EF6 ■-本部分中所示的 api 在实体框架6中引入。如果你使用的是早期版本, 则不会应用此信息。

生成视图

用于生成视图的 Api 位于 StorageMappingItemCollection 类中。""。可以使用 ObjectContext 的 MetadataWorkspace 检索上下文的 StorageMappingCollection。如果你使用的是较新的 DbContext API, 则可以使用如下所示的 IObjectContextAdapter 进行访问:在此代码中, 我们有一个名为 dbContext 的派生 DbContext 实例:

```
var objectContext = ((IObjectContextAdapter) dbContext).ObjectContext;
var mappingCollection = (StorageMappingItemCollection)objectContext.MetadataWorkspace
    .GetItemCollection(DataSpace.CSSpace);
```

获得 StorageMappingItemCollection 后, 可以访问 GenerateViews 和 ComputeMappingHashValue 方法。

```
public Dictionary<EntitySetBase, DbMappingView> GenerateViews(IList<EdmSchemaError> errors)
public string ComputeMappingHashValue()
```

第一种方法为容器映射中的每个视图创建一个包含条目的字典。第二种方法为单个容器映射计算哈希值, 并在运行时使用该方法验证模型是否自生成视图以来未发生更改。对于涉及多个容器映射的复杂方案, 会提供两种方法的替代。

生成视图时, 将调用 GenerateViews 方法, 然后写出生成的 EntitySetBase 和 DbMappingView。还需要存储 ComputeMappingHashValue 方法生成的哈希。

正在加载视图

为了加载 GenerateViews 方法生成的视图, 你可以为 EF 提供从 DbMappingViewCache 抽象类继承的类。DbMappingViewCache 指定必须实现的两个方法:

```
public abstract string MappingHashValue { get; }
public abstract DbMappingView GetView(EntitySetBase extent);
```

MappingHashValue 属性必须返回 ComputeMappingHashValue 方法生成的哈希值。当 EF 打算请求视图时, 它将首先生成并将模型的哈希值与此属性返回的哈希值进行比较。如果二者不匹配, 则 EF 将引发 EntityCommandCompilationException 异常。

GetView 方法将接受 EntitySetBase, 并且需要返回一个 DbMappingView, 其中包含为与在 EntitySetBase 方法生成的字典中的给定 GenerateViews 关联的 EntitySql。如果 EF 要求你提供不具有的视图, 则 GetView 应返回 null。

下面是 DbMappingViewCache 中的一种提取功能, 如上文所述, 使用 Power Tools 生成, 其中有一种方法可以存储和检索所需的 EntitySql。

```

public override string MappingHashValue
{
    get { return "a0b843f03dd29abee99789e190a6fb70ce8e93dc97945d437d9a58fb8e2af2e"; }
}

public override DbMappingView GetView(EntitySetBase extent)
{
    if (extent == null)
    {
        throw new ArgumentNullException("extent");
    }

    var extentName = extent.EntityContainer.Name + "." + extent.Name;

    if (extentName == "BlogContext.Blogs")
    {
        return GetView2();
    }

    if (extentName == "BlogContext.Posts")
    {
        return GetView3();
    }

    return null;
}

private static DbMappingView GetView2()
{
    return new DbMappingView(@"
        SELECT VALUE -- Constructing Blogs
        [BlogApp.Models.Blog](T1.Blog_BlogId, T1.Blog_Test, T1.Blog_title, T1.Blog_Active,
        T1.Blog_SomeDecimal)
        FROM (
        SELECT
            T.BlogId AS Blog_BlogId,
            T.Test AS Blog_Test,
            T.title AS Blog_title,
            T.Active AS Blog_Active,
            T.SomeDecimal AS Blog_SomeDecimal,
            True AS _from0
        FROM CodeFirstDatabase.Blog AS T
        ) AS T1");
}

```

若要让 EF 使用 `DbMappingViewCache`, 请使用 `DbMappingViewCacheTypeAttribute`, 同时指定为其创建的上下文。在下面的代码中, 我们将 `BlogContext` 与 `MyMapViewCache` 类相关联。

```
[assembly: DbMappingViewCacheType(typeof(BlogContext), typeof(MyMapViewCache))]
```

对于更复杂的方案, 可以通过指定映射视图缓存工厂来提供映射视图缓存实例。此操作可通过实现抽象类 `MappingViews.DbMappingViewCacheFactory` 来完成。可以使用 `StorageMappingItemCollection.MappingViewCacheFactoryProperty` 检索或设置所使用的映射视图缓存工厂的实例。

实体框架 6 提供程序

2020/4/8 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

实体框架现在正在开源许可证下开发，EF6 及更高版本不会作为 .NET Framework 的一部分提供。这样做有许多好处，但也需要针对 EF6 程序集重建 EF 提供程序。这就意味着 EF5 及以下版本的 EF 提供程序在重建之前不能用于 EF6。

哪些提供程序可用于 EF6？

已知的针对 EF6 重建的提供程序包括：

- Microsoft SQL Server 提供程序
 - 构建于[实体框架开源代码库](#)
 - 作为[EntityFramework NuGet 包](#)的一部分提供
- Microsoft SQL Server Compact Edition 提供程序
 - 构建于[实体框架开源代码库](#)
 - 在[EntityFramework.SqlServerCompact NuGet 包](#)中提供
- Devart dotConnect 数据提供程序
 - 来自[Devart](#)的提供各种数据库的第三方提供程序，包括 Oracle、MySQL、PostgreSQL、SQLite、Salesforce、DB2 和 SQL Server
- CData Software 提供程序
 - 来自[CData Software](#)的提供各种数据存储的第三方提供程序，包括 Salesforce、Azure 表存储、MySQL 等等
- Firebird 提供程序
 - 作为[NuGet 包](#)提供
- Visual Fox Pro 提供程序
 - 作为[NuGet 包](#)提供
- MySQL
 - [MySQL Connector/NET for Entity Framework](#)
- PostgreSQL
 - Npgsql 作为[NuGet 包](#)提供
- Oracle
 - ODP.NET 作为[NuGet 包](#)提供

请注意，此列表并不表示给定提供程序的功能级别或支持情况，只表示已构建了 EF6。

注册 EF 提供程序

从实体框架 6 开始，可以使用基于代码的配置或在应用程序的配置文件中注册 EF 提供程序。

配置文件注册

在 app.config 或 web.config 中注册 EF 提供程序的格式如下：

```
<entityFramework>
  <providers>
    <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
  </providers>
</entityFramework>
```

请注意，如果从 NuGet 安装 EF 提供程序，则通常 NuGet 包会自动将此注册添加到配置文件中。如果向非应用启动项目安装 NuGet 包，则可能需要将该注册复制到启动项目的配置文件中。

此注册中的“invariantName”是用于标识 ADO.NET 提供程序的相同固定名称。它是 DbProviderFactories 注册中的“invariant”属性，是连接字符串注册中的“providerName”属性。要使用的固定名称也应包含在该提供程序的文档中。固定名称的示例：SQL Server 为“System.Data.SqlClient”，SQL Server Compact 为“System.Data.SqlServerCe.4.0”。

此注册中的“类型”是派生自“System.Data.Entity.Core.Common.DbProviderServices”的提供程序类型的程序集限定名称。例如，用于 SQL Compact 的字符串是“System.Data.Entity.SqlServerCompact.SqlCeProviderServices，EntityFramework.SqlServerCompact”。此处要使用的类型应包含在该提供程序的文档中。

基于代码的注册

从实体框架 6 开始，可在代码中指定整个应用程序的 EF 配置。有关完整的详细信息，请参阅[实体框架基于代码的配置](#)。使用基于代码的配置注册 EF 提供程序的常规方法是，创建一个派生自 System.Data.Entity.DbConfiguration 的新类，并将其放置在与 DbContext 类相同的程序集中。然后，DbConfiguration 类应在其构造函数中注册该提供程序。例如，要注册 SQL Compact 提供程序，DbConfiguration 类如下所示：

```
public class MyConfiguration : DbConfiguration
{
    public MyConfiguration()
    {
        SetProviderServices(
            SqlCeProviderServices.ProviderInvariantName,
            SqlCeProviderServices.Instance);
    }
}
```

在此代码中，“SqlCeProviderServices.ProviderInvariantName”为 SQL Server Compact 提供程序的固定名称字符串（“System.Data.SqlServerCe.4.0”）提供了便利，SqlCeProviderServices.Instance 会返回 SQL Compact EF 提供程序的单一实例。

如果未提供我需要的提供程序怎么办？

如果该提供程序适用于以前版本的 EF，则建议联系提供程序的所有者并要求他们创建 EF6 版本。应包含对[EF6 提供程序模型的文档](#)的引用。

可以自己编写提供程序吗？

当然可以自己创建 EF 提供程序，但不应把这项任务考虑的太简单。可以从上面关于 EF6 提供程序模型的链接开始着手。你可能还会发现，从[EF 开源代码库](#)中包含的 SQL Server 和 SQL CE 提供程序的代码着手或用作参考会很有用。

请注意，从 EF6 开始，EF 提供程序与基础 ADO.NET 提供程序的耦合紧密程度降低。这使得编写 EF 提供程序更轻松，且无需编写或包装 ADO.NET 类。

实体框架6提供程序模型

2020/3/11 •

实体框架提供程序模型允许实体框架与不同类型的数据库服务器一起使用。例如，可以插入一个提供程序以允许对 Microsoft SQL Server 使用 EF，而另一个提供程序可以插入到中，以允许在 Microsoft SQL Server Compact 版本中使用 EF。[实体框架提供程序](#)页上提供了可识别的 EF6 提供程序。

EF 与提供程序交互的方式需要进行某些更改，以允许在开源许可证下释放 EF。这些更改要求针对 EF6 程序集重新生成 EF 提供程序，同时提供用于注册提供程序的新机制。

重建

在 EF6 中，先前 .NET Framework 的核心代码现在作为带外 (OOB) 程序集发货。有关如何针对 EF6 生成应用程序的详细信息，请参阅[更新应用程序的 EF6](#)页。还需要使用这些说明重新生成提供程序。

提供程序类型概述

EF 提供程序实际上是由 CLR 类型定义的特定于提供程序的服务的集合，这些服务由这些服务扩展（用于基类）或实现（用于接口）。其中的两个服务都是基本的功能，而 EF 是必需的。其他选项是可选的，仅在需要特定功能时才需要实现，并且/或者这些服务的默认实现对于目标特定数据库服务器不起作用。

基本提供程序类型

DbProviderFactory

EF 依赖于从[DbProviderFactory](#)派生的类型，以执行所有低级别的数据库访问。DbProviderFactory 实际上不是 EF 的一部分，而是 .NET Framework 中的一个类，它为 ADO.NET 提供程序提供了一个入口点，这些提供程序可由 EF、其他 O/RMs 或应用程序直接使用以获取连接的实例、命令、参数和其他 ADO.NET 抽象的实例。有关 DbProviderFactory 的详细信息，请参阅[ADO.NET 的 MSDN 文档](#)。

DbProviderServices

EF 依赖于从 Dbproviderservices.createdatabase 派生的类型，以便在 ADO.NET 提供程序已提供的功能之上提供 EF 所需的其他功能。在 EF 的较早版本中，Dbproviderservices.createdatabase 类是 .NET Framework 的一部分，并且在 System.web 命名空间中找到。从 EF6 开始，此类现在是 EntityFramework 的一部分，并且位于命名空间中。

可在[MSDN](#)上找到有关 dbproviderservices.createdatabase 实现的基本功能的更多详细信息。但请注意，在撰写本文时，尽管大部分概念仍然有效，但不会对 EF6 进行更新。Dbproviderservices.createdatabase 的 SQL Server 和 SQL Server Compact 实现也签入到[开源基本代码](#)，并可用作其他实现的有用引用。

在早期版本的 EF 中，Dbproviderservices.createdatabase 实现直接从 ADO.NET 提供程序获取。这是通过将 DbProviderFactory 强制转换为 IServiceProvider 并调用 GetService 方法来完成的。这会将 EF 提供程序紧密耦合到 DbProviderFactory。此耦合阻止的 EF 被移出 .NET Framework，因此对于 EF6，已删除此紧密耦合，并且 Dbproviderservices.createdatabase 的实现现在直接在应用程序的配置文件或基于代码的配置中注册，如下面的_注册 dbproviderservices.createdatabase_部分中所述。

其他服务

除了上面所述的基本服务外，EF 还使用许多其他服务，它们始终或有时是特定于提供程序的服务。这些服务的默认特定于提供程序的实现可由 Dbproviderservices.createdatabase 实现提供。应用程序还可以覆盖这些服务的实现，或在 Dbproviderservices.createdatabase 类型不提供默认值时提供实现。下面更详细地介绍了_这一情

况_。

下面列出了提供程序可能对提供程序感兴趣的其他服务类型。有关每种服务类型的更多详细信息，请参阅 API 文档。

IDbExecutionStrategy

这是一种可选的服务，当对数据库执行查询和命令时，此服务允许提供程序实现重试或其他行为。如果未提供实现，则 EF 只会执行命令并传播引发的任何异常。对于 SQL Server 此服务用于提供重试策略，这在对基于云的数据库服务器(如 SQL Azure)运行时特别有用。

IDbConnectionFactory

这是一种可选服务，允许提供程序在只给定数据库名称的情况下按约定创建 DbConnection 对象。请注意，虽然此服务可以通过 Dbproviderservices.createdatabase 实现进行解析，但它已存在于 EF 4.1，并且还可以在配置文件或代码中显式设置。如果已将此服务注册为默认提供程序(请参阅下面_的默认提供程序_)，并且未在其他位置设置默认的连接工厂，则提供程序将只有机会解析此服务。

DbSpatialServices

这是一种可选服务，允许提供商添加对地理空间类型和几何空间类型的支持。必须提供此服务的实现，应用程序才能将 EF 与空间类型一起使用。将以两种方式请求 DbSptialServices。首先，使用 DbProviderInfo 对象(其中包含固定名称和清单标记)作为密钥请求提供程序特定的空间服务。其次，无关键字就可以要求 DbSpatialServices。这用于解析在创建独立的 DbGeography 或 DbGeometry 类型时使用的 "全局空间提供程序"。

MigrationSqlGenerator

这是一项可选的服务，它允许将 EF 迁移用于通过 Code First 创建和修改数据库架构时使用的 SQL 生成。需要实现才能支持迁移。如果提供了实现，则在使用数据库初始值设定项或 Database.Create 方法创建数据库时，也将使用它。

Func < DbConnection, string, HistoryContextFactory >

这是一种可选的服务，它允许提供程序配置 HistoryContext 与 EF 迁移使用的 `_MigrationHistory` 表的映射。HistoryContext 是一个 Code First DbContext，可以使用普通 Fluent API 进行配置，以更改表名称和列映射规范等项。如果该提供程序支持所有默认表和列映射，则由 EF 为所有提供程序返回的此服务的默认实现可能适用于给定的数据库服务器。在这种情况下，提供程序不需要提供此服务的实现。

IDbProviderFactoryResolver

这是一种可选的服务，用于从给定的 DbConnection 对象中获取正确的 DbProviderFactory。此服务的默认实现由 EF 为所有提供程序返回，旨在适用于所有提供程序。但是，在 .NET 4 上运行时，如果 DbProviderFactory 的 DbConnections，则无法从一个公共访问。因此，EF 使用一些试探法搜索已注册的提供程序以找到匹配项。某些提供程序的这种试探方法可能会失败，并且在这种情况下，提供程序应提供新的实现。

注册 Dbproviderservices.createdatabase

要使用的 Dbproviderservices.createdatabase 实现可在应用程序的配置文件(app.config 或 web.config)或使用基于代码的配置中注册。在任一情况下，注册都使用提供程序的 "固定名称" 作为密钥。这允许在单个应用程序中注册和使用多个提供程序。用于 EF 注册的固定名称与用于 ADO.NET 提供程序注册和连接字符串的固定名称相同。例如，对于 SQL Server 使用固定名称 "SqlClient"。

配置文件注册

要使用的 Dbproviderservices.createdatabase 类型在应用程序配置文件的 entityFramework 节的提供程序列表中注册为提供程序元素。例如：

```
<entityFramework>
  <providers>
    <provider invariantName="My.Invariant.Name" type="MyProvider.MyProviderServices, MyAssembly" />
  </providers>
</entityFramework>
```

_类型_字符串必须是要使用的 dbproviderservices.createdatabase 实现的程序集限定的类型名称。

基于代码的注册

从 EF6 提供程序开始, 还可以使用代码注册。这允许使用 EF 提供程序, 而无需对应用程序的配置文件进行任何更改。若要使用基于代码的配置, 应用程序应按照[基于代码的配置文档](#)中所述创建 DbConfiguration 类。然后, DbConfiguration 类的构造函数应调用 SetProviderServices 来注册 EF 提供程序。例如:

```
public class MyConfiguration : DbConfiguration
{
  public MyConfiguration()
  {
    SetProviderServices("My.New.Provider", new MyProviderServices());
  }
}
```

解析其他服务

如前所述, 在 "提供程序类型概述" 部分中, 还可以使用 dbproviderservices.createdatabase 类来解析其他服务。这是可能的, 因为 Dbproviderservices.createdatabase 实现了 IDbDependencyResolver, 并且每个注册的 Dbproviderservices.createdatabase 类型都添加为 "默认冲突解决程序"。[依赖项解析](#)中更详细地介绍了 IDbDependencyResolver 机制。但是, 不需要了解此规范中的所有概念来解析提供程序中的其他服务。

提供程序解析附加服务的最常见方法是为 Dbproviderservices.createdatabase 类的构造函数中的每个服务调用 Dbproviderservices.createdatabase。例如, SqlProviderServices (用于 SQL Server 的 EF 提供程序) 具有类似于以下内容的代码:

```
private SqlProviderServices()
{
  AddDependencyResolver(new SingletonDependencyResolver<IDbConnectionFactory>(
    new SqlConnectionFactory()));

  AddDependencyResolver(new ExecutionStrategyResolver<DefaultSqlExecutionStrategy>(
    "System.data.SqlClient", null, () => new DefaultSqlExecutionStrategy()));

  AddDependencyResolver(new SingletonDependencyResolver<Func<MigrationSqlGenerator>>(
    () => new SqlServerMigrationSqlGenerator(), "System.data.SqlClient"));

  AddDependencyResolver(new SingletonDependencyResolver<DbSpatialServices>(
    SqlSpatialServices.Instance,
    k =>
    {
      var asSpatialKey = k as DbProviderInfo;
      return asSpatialKey == null
        || asSpatialKey.ProviderInvariantName == ProviderInvariantName;
    }));
}
```

此构造函数使用以下帮助器类:

- `SingletonDependencyResolver`: 提供一种解析单独服务的简单方法, 即每次调用 `GetService` 时都为其返回同一个实例的服务。暂时性服务通常注册为一个单一实例, 将用于按需创建暂时性实例。
- `ExecutionStrategyResolver`: 特定于返回 `IExecutionStrategy` 实现的解析程序。

还可以重写 `Dbproviderservices.createdatabase` 并直接解析附加服务，而不是使用 `Dbproviderservices.createdatabase`。`AddDependencyResolver`。当 EF 需要由特定类型定义的服务时，将调用此方法，在某些情况下，将调用此方法作为给定的键。如果服务可以返回服务，则该方法应返回它，或者返回 `null` 以选择退出该服务，而另一个类可以解决它。例如，若要解析默认的连接工厂，`GetService` 中的代码可能如下所示：

```
public override object GetService(Type type, object key)
{
    if (type == typeof(IDbConnectionFactory))
    {
        return new SqlConnectionFactory();
    }
    return null;
}
```

注册顺序

在应用程序的配置文件中注册多个 `Dbproviderservices.createdatabase` 实现时，它们将按照它们列出的顺序添加为辅助解析程序。由于解析程序始终添加到辅助解析程序链的顶部，这意味着，列表末尾的提供程序将有机会解析其他依赖项。（这在最初看起来可能比较直观，但如果你想象你将每个提供程序移出列表，并将其堆积在现有的提供商之上，这会很有帮助。）

此顺序通常并不重要，因为大多数提供程序服务都是特定于提供程序的，并使用提供程序固定名称进行键控。但是，对于不是由提供程序固定名称或其他特定于提供程序的密钥进行键控的服务，将根据此顺序解析服务。例如，如果未在其他任何位置以不同方式显式设置，则默认连接工厂将来自链中最顶层的提供程序。

其他配置文件注册

可以直接在应用程序的配置文件中直接注册上述部分所述的其他提供程序服务。完成此操作后，将使用配置文件中的注册，而不是 `Dbproviderservices.createdatabase` 实现的 `GetService` 方法返回的任何内容。

注册默认的连接工厂

从 EF5 开始，`EntityFramework` NuGet 包自动在配置文件中注册 SQL Express 连接工厂或 LocalDb 连接工厂。

例如：

```
<entityFramework>
  <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">
  </defaultConnectionFactory>
</entityFramework>
```

`_类型_` 为默认连接工厂的程序集限定类型名称，该名称必须实现 `IDbConnectionFactory`。

建议提供程序 NuGet 包在安装时以这种方式设置默认连接工厂。请参阅下面_的提供程序的 NuGet 包_。

其他 EF6 提供程序更改

空间提供程序更改

支持空间类型的提供程序现在必须在派生自 `DbSpatialDataReader` 的类上实现一些附加方法：

- `public abstract bool IsGeographyColumn(int ordinal)`
- `public abstract bool IsGeometryColumn(int ordinal)`

此外，还提供了一些新的异步版本，建议将它们作为默认的实现委托给同步方法，并因此不会以异步方式执行：

- `public virtual Task<DbGeography> GetGeographyAsync(int ordinal, CancellationToken cancellationToken)`
- `public virtual Task<DbGeometry> GetGeometryAsync(int ordinal, CancellationToken cancellationToken)`

对可枚举的本机支持。包含

EF6 引入了一个新的表达式类型 `DbInExpression`, 此类型已添加到解决有关使用可枚举的性能问题。包含在 LINQ 查询中。`DbProviderManifest` 类具有新的虚方法 `SupportsInExpression`, 该方法由 EF 调用来确定提供程序是否处理新的表达式类型。为了与现有提供程序实现兼容, 此方法返回 `false`。为了从此改进中获益, EF6 提供程序可添加代码来处理 `DbInExpression`, 并覆盖 `SupportsInExpression` 以返回 `true`。可以通过调用 `DbExpressionBuilder.In` 方法创建 `DbInExpression` 的实例。`DbInExpression` 实例由 `DbExpression` 组成, 通常表示表列, 以及用于测试匹配项的 `DbConstantExpression` 列表。

提供程序的 NuGet 包

使 EF6 提供程序可用的一种方法是将其发布为 NuGet 包。使用 NuGet 包具有以下优势:

- 可以轻松地将提供程序注册到应用程序的配置文件中, 使用 NuGet
- 可以对配置文件进行其他更改, 以设置默认的连接工厂, 以便约定建立的连接将使用已注册的提供程序
- NuGet 处理添加绑定重定向, 以便 EF6 提供程序在新的 EF 包发布后仍可继续工作

下面是一个包含在[开源代码库](#)中的 `EntityFramework.SqlServerCompact` 包。此包提供了一个用于创建 EF 提供程序 NuGet 包的好模板。

PowerShell 命令

安装 `EntityFramework` NuGet 包时, 它会注册一个包含两个命令的 PowerShell 模块, 这些命令对于提供程序包非常有用:

- `EFProvider` 在目标项目的配置文件中为该提供程序添加了一个新的实体, 并确保其位于已注册提供程序列表的末尾。
- `EFDynamicConnectionFactory` 添加或更新目标项目的配置文件中的 `defaultConnectionFactory` 注册。

这两个命令都负责将 `entityFramework` 节添加到配置文件中, 并在必要时添加提供程序集合。

这是为了从安装.ps1 NuGet 脚本中调用这些命令。例如, 安装.ps1 for SQL Compact 提供程序如下所示:

```
param($installPath, $toolsPath, $package, $project)
Add-EFDefaultConnectionFactory $project 'System.Data.Entity.Infrastructure.SqlCeConnectionFactory,
EntityFramework' -ConstructorArguments 'System.Data.SqlClient.4.0'
Add-EFProvider $project 'System.Data.SqlClient.4.0'
'System.Data.Entity.SqlClient.SqlCeProviderServices, EntityFramework.SqlServerCompact'</pre>
```

可以在“包管理器控制台”窗口中使用 `get-help` 获取有关这些命令的详细信息。

包装提供程序

包装提供程序是一个 EF 和/或 ADO.NET 提供程序, 它包装现有提供程序以使用其他功能(如分析或跟踪功能)对其进行扩展。包装提供程序可以通过正常方式进行注册, 但在运行时通过拦截与提供程序相关的服务的解决方法, 在运行时设置包装提供程序通常更为方便。`DbConfiguration` 类上的静态事件 `OnLockingConfiguration` 可用于执行此操作。

在 EF 确定了将从其获取应用程序域的所有 EF 配置之后, 但在锁定以供使用之前, 将调用 `OnLockingConfiguration`。在应用启动时(使用 EF 之前), 应用应为此事件注册事件处理程序。(我们正在考虑在配置文件中添加对注册此处理程序的支持, 但目前尚不支持此项。)然后, 事件处理程序应为需要包装的每个服务调用 `ReplaceService`。

例如, 若要将 `IDbConnectionFactory` 和 `DbProviderService` 进行包装, 应注册类似于以下内容的处理程序:

```
DbConfiguration.OnLockingConfiguration +=
    (_ , a) =>
{
    a.ReplaceService<DbProviderServices>(
        (s, k) => new MyWrappedProviderServices(s));

    a.ReplaceService<IDbConnectionFactory>(
        (s, k) => new MyWrappedConnectionFactory(s));
};
```

已解决的服务现在应与用于解析服务的密钥一起打包到处理程序。然后，处理程序可以包装此服务，并将返回的服务替换为包装的版本。

使用 EF 解析 DbProviderFactory

DbProviderFactory 是 EF 所需的一种基本提供程序类型，如上面的 "提供程序类型概述" 部分所述。如上所述，它不是 EF 类型，注册通常不是 EF 配置的一部分，而是在 machine.config 文件和/或应用程序的配置文件中注册的正常 ADO.NET 提供程序。

尽管在查找要使用的 DbProviderFactory 时，此 EF 仍使用其常规依赖项解析机制。默认冲突解决程序使用配置文件中的常规 ADO.NET 注册，因此这通常是透明的。但由于使用的是正常的依赖项解析机制，这意味着即使未完成普通 ADO.NET 注册，也可以使用 IDbDependencyResolver 来解析 DbProviderFactory。

以这种方式解析 DbProviderFactory 有几个含义：

- 使用基于代码的配置的应用程序可在其 DbConfiguration 类中添加调用来注册相应的 DbProviderFactory。这对于不希望（或无法）使用任何基于文件的配置的应用程序特别有用。
- 该服务可以使用 ReplaceService 进行包装或替换，如以上_包装提供商_部分所述
- 理论上，Dbproviderservices.createdatabase 实现可以解析 DbProviderFactory。

执行以上任一操作时需要注意的重要一点是，它们只会影响按 EF 查找 DbProviderFactory。其他非 EF 代码可能仍会预期 ADO.NET 提供程序以正常方式注册，如果找不到注册，可能会失败。出于此原因，通过正常的 ADO.NET 方式注册 DbProviderFactory 通常更好。

相关服务

如果使用 EF 解析 DbProviderFactory，则它还应该解析 IProviderInvariantName 和 IDbProviderFactoryResolver 服务。

IProviderInvariantName 是一种服务，用于确定给定类型 DbProviderFactory 的提供程序固定名称。此服务的默认实现使用 ADO.NET 提供程序注册。这意味着，如果 ADO.NET 提供程序未以正常方式注册，因为 DbProviderFactory 正在由 EF 解析，则还需要解析此服务。请注意，使用 DbConfiguration. SetProviderFactory 方法时，会自动添加此服务的解析程序。

如上面的 "提供程序类型概述" 部分所述， IDbProviderFactoryResolver 用于从给定的 DbConnection 对象获取正确的 DbProviderFactory。此服务在 .NET 4 上运行时的默认实现使用 ADO.NET 提供程序注册。这意味着，如果 ADO.NET 提供程序未以正常方式注册，因为 DbProviderFactory 正在由 EF 解析，则还需要解析此服务。

提供程序对空间类型的支持

2020/3/11 ·

实体框架支持通过 `DbGeography` 或 `DbGeometry` 类处理空间数据。这些类依赖于由实体框架提供程序提供的数据库特定功能。并非所有提供程序都支持空间数据，其中可能有其他先决条件，如安装空间类型程序集。下面提供了有关空间类型提供程序支持的详细信息。

有关如何在应用程序中使用空间类型的其他信息，请参阅两个演练，一个用于 `Code First`，另一个用于 `Database First` 或 `Model First`：

- [Code First 中的空间数据类型](#)
- [EF 设计器中的空间数据类型](#)

支持空间类型的 EF 版本

EF5 中引入了对空间类型的 support。但是，仅当应用程序在 .NET 4.5 上面向和运行时，才支持 EF5 空间类型。

对于面向 .NET 4 和 .NET 4.5 的应用程序，支持从 EF6 空间类型开始。

支持空间类型的 EF 提供程序

EF5

我们意识到支持空间类型的 EF5 的实体框架提供程序是：

- Microsoft SQL Server 提供程序
 - 此提供程序作为 EF5 的一部分提供。
 - 此提供程序依赖于一些可能需要安装的其他低级别库，请参阅下面的详细信息。
- [适用于 Oracle 的 Devart dotConnect](#)
 - 这是 Devart 中的第三方提供程序。

如果你知道支持空间类型的 EF5 提供程序，请联系，我们将很高兴地将其添加到此列表。

EF6

我们意识到支持空间类型的 EF6 的实体框架提供程序是：

- Microsoft SQL Server 提供程序
 - 此提供程序作为 EF6 的一部分提供。
 - 此提供程序依赖于一些可能需要安装的其他低级别库，请参阅下面的详细信息。
- [适用于 Oracle 的 Devart dotConnect](#)
 - 这是 Devart 中的第三方提供程序。

如果你知道支持空间类型的 EF6 提供程序，请联系，我们将很高兴地将其添加到此列表。

具有 Microsoft SQL Server 的空间类型的先决条件

SQL Server 空间支持取决于低级别、SQL Server 特定类型 `SqlGeography` 和 `SqlGeometry`。这些类型存在于 `mscorlib.dll` 程序集中，并且此程序集不作为 EF 的一部分或作为 .NET Framework 的一部分。

安装 Visual Studio 时，它通常还会安装 SQL Server 的版本，这将包括安装 Microsoft。

如果 SQL Server 未安装在要使用空间类型的计算机上，或者如果从 SQL Server 安装中排除了空间类型，则需要手动安装它们。可以使用 [`SQLSysClrTypes.msi`](#)（这是 Microsoft SQL Server 功能包的一部分）安装类型。空间类型

SQL Server 特定于版本，因此我们建议在 Microsoft 下载中心[搜索 "SQL Server 功能包"](#)，然后选择和下载与将使用的 SQL Server 版本相对应的选项。

使用代理

2020/3/11 ·

创建 POCO 实体类型实例时，实体框架通常会创建动态生成的派生类型的实例，该类型充当实体的代理。此代理会重写实体的某些虚拟属性，以插入挂钩，以便在访问属性时自动执行操作。例如，此机制用于支持关系的延迟加载。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

禁用代理创建

有时，阻止实体框架创建代理实例会很有用。例如，序列化非代理实例比序列化代理实例要容易得多。可以通过清除 `ProxyCreationEnabled` 标志关闭代理创建。您可以在上下文的构造函数中执行此操作。例如：

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.ProxyCreationEnabled = false;
    }

    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

请注意，EF 不会为代理创建没有的类型的代理。这意味着，你还可以通过使用密封的类型和/或没有虚拟属性来避免代理。

显式创建代理的实例

如果使用 `new` 运算符创建实体的实例，则不会创建代理实例。这可能不是问题，但如果需要创建代理实例（例如，使延迟加载或代理更改跟踪有效），则可以使用 `DbSet` 的 `Create` 方法来执行此操作。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Create();
}
```

如果要创建派生实体类型的实例，则可以使用 `Create` 的泛型版本。例如：

```
using (var context = new BloggingContext())
{
    var admin = context.Users.Create<Administrator>();
}
```

请注意，`Create` 方法不会将创建的实体添加或附加到上下文。

请注意，如果为实体创建代理类型，`Create` 方法将仅创建实体类型本身的实例，因为它不会执行任何操作。例如，如果实体类型是密封的并且/或没有虚拟属性，则 `Create` 将只创建实体类型的实例。

从代理类型获取实际实体类型

代理类型的名称如下所示：

可以使用 `ObjectContext` 中的 `GetObjectType` 方法查找此代理类型的实体类型。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    var entityType = ObjectContext.GetObjectType(blog.GetType());
}
```

请注意，如果传递给 `GetObjectType` 的类型是不是代理类型的实体类型的实例，则仍将返回实体的类型。这意味着，始终可以使用此方法获取实际的实体类型，而无需进行任何其他检查即可查看该类型是否为代理类型。

使用模拟框架进行测试

2020/3/11 ·

NOTE

- EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

编写应用程序测试时，通常需要避免数据库的命中。实体框架使你可以通过创建上下文来实现此目的，该上下文具有测试定义的行为，从而利用内存中数据。

用于创建测试双精度的选项

可使用两种不同的方法创建上下文的内存中版本。

- **创建自己的测试双精度型**-此方法涉及编写您自己的上下文和 dbset 的内存中实现。这使你可以很好地控制类的行为，但可能涉及到编写和拥有合理的代码量。
- **使用模拟框架创建测试双精度型**-使用模拟框架(如 Moq)，可以在运行时为你的上下文提供内存中实现并在运行时动态创建。

本文将介绍如何使用模拟框架。若要创建自己的测试，请参阅[采用自己的测试进行测试双精度](#)。

为了演示如何将 EF 与模拟框架结合使用，我们将使用 Moq。获取 Moq 的最简单方法是[从 NuGet 安装 Moq 包](#)。

用预 EF6 版本测试

本文中所述的方案取决于我们对 EF6 中的 DbSet 进行的一些更改。若要使用 EF5 和更早的版本进行测试，请参阅[使用虚设上下文进行测试](#)。

EF 内存中测试的限制双精度

内存中测试双精度型可以是一种提供使用 EF 的应用程序的单元测试级别的好方法。但是，在执行此操作时，将使用 LINQ to Objects 对内存中数据执行查询。这可能会导致不同的行为，而不是使用 EF 的 LINQ 提供程序(LINQ to Entities)将查询转换为对数据库运行的 SQL。

这种差异的一个示例就是加载相关数据。如果创建一系列博客，其中每个博客都有相关的文章，则在使用内存中数据时，将始终为每个博客加载相关文章。但是，在对数据库运行时，仅当使用 Include 方法时才会加载数据。

出于此原因，建议始终包括某个级别的端到端测试(除了单元测试)，以确保应用程序能够对数据库正常运行。

与本文一起介绍

本文提供了完整的代码清单，你可以将其复制到 Visual Studio 中，以根据你的需要进行。最简单的方法是创建一个单元测试项目，并需要将 .NET Framework 4.5 作为目标来完成使用 async 的部分。

EF 模型

要测试的服务利用的是由 "Bloggingcontext" 和博客和 Post 类组成的 EF 模型。此代码可能是由 EF 设计器生成的，或是 Code First 模型中。

```

using System.Collections.Generic;
using System.Data.Entity;

namespace TestingDemo
{
    public class BloggingContext : DbContext
    {
        public virtual DbSet<Blog> Blogs { get; set; }
        public virtual DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}

```

虚拟 DbSet 属性与 EF 设计器

请注意，上下文中的 DbSet 属性标记为虚拟。这将允许模拟框架从我们的上下文派生，并使用模拟实现覆盖这些属性。

如果使用 Code First，则可以直接编辑类。如果使用的是 EF 设计器，则需要编辑生成上下文的 T4 模板。打开 <model_name>.Context.tt 文件嵌套在 edmx 文件下，查找以下代码片段，并将其添加到 virtual 关键字中，如下所示。

```

public string DbSet(EntitySet entitySet)
{
    return string.Format(
        CultureInfo.InvariantCulture,
        "{0} virtual DbSet\<{1}> {2} {{ get; set; }}",
        Accessibility.ForReadOnlyProperty(entitySet),
        _typeMapper.GetTypeName(entitySet.ElementType),
        _code.Escape(entitySet));
}

```

要测试的服务

为了演示使用内存中测试的测试，我们将为 BlogService 编写一些测试。该服务可以创建新的博客（AddBlog）并返回按名称排序的所有博客（GetAllBlogs）。除了 GetAllBlogs 之外，我们还提供了一个方法，该方法将异步获取按名称（GetAllBlogsAsync）排序的所有博客。

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class BlogService
    {
        private BloggingContext _context;

        public BlogService(BloggingContext context)
        {
            _context = context;
        }

        public Blog AddBlog(string name, string url)
        {
            var blog = _context.Blogs.Add(new Blog { Name = name, Url = url });
            _context.SaveChanges();

            return blog;
        }

        public List<Blog> GetAllBlogs()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return query.ToList();
        }

        public async Task<List<Blog>> GetAllBlogsAsync()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return await query.ToListAsync();
        }
    }
}

```

测试非查询方案

这就是开始测试非查询方法所需的所有操作。以下测试使用 Moq 创建上下文。然后，它将创建一个 DbSet<博客> 并向其线路，使其从上下文的博客属性返回。接下来，使用上下文创建新的 BlogService，然后使用 AddBlog 方法 创建新的博客。最后，测试将验证该服务是否在上下文中添加了新的博客并调用了 SaveChanges。

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Data.Entity;

namespace TestingDemo
{
    [TestClass]
    public class NonQueryTests
    {
        [TestMethod]
        public void CreateBlog_saves_a_blog_via_context()
        {
            var mockSet = new Mock<DbSet<Blog>>();

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(m => m.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            service.AddBlog("ADO.NET Blog", "http://blogs.msdn.com/adonet");

            mockSet.Verify(m => m.Add(It.IsAny<Blog>()), Times.Once());
            mockContext.Verify(m => m.SaveChanges(), Times.Once());
        }
    }
}
```

测试查询方案

为了能够对 DbSet 测试执行查询，我们需要设置 IQueryble 的实现。第一步是创建一些内存中数据，我们使用的是博客><列表。接下来，我们将创建一个上下文和 DBSet<博客> 然后将 DbSet 的 IQueryble 实现连接在一起—它们只是委托给适用于 List<T> 的 LINQ to Objects 提供程序。

然后，可以根据我们的测试来创建 BlogService，并确保从 GetAllBlogs 返回的数据按名称排序。

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

namespace TestingDemo
{
    [TestClass]
    public class QueryTests
    {
        [TestMethod]
        public void GetAllBlogs_orders_by_name()
        {
            var data = new List<Blog>
            {
                new Blog { Name = "BBB" },
                new Blog { Name = "ZZZ" },
                new Blog { Name = "AAA" },
            }.AsQueryable();

            var mockSet = new Mock<DbSet<Blog>>();
            mockSet.As<IQueryable<Blog>>().Setup(m => m.Provider).Returns(data.Provider);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.Expression).Returns(data.Expression);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.ElementType).Returns(data.ElementType);
            mockSet.As<IQueryable<Blog>>().Setup(m => m.GetEnumerator()).Returns(data.GetEnumerator());

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(c => c.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            var blogs = service.GetAllBlogs();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

用异步查询进行测试

实体框架6引入了一组扩展方法，这些方法可用于以异步方式执行查询。这些方法的示例包括 `ToListAsync`、`FirstAsync`、`ForEachAsync` 等。

由于实体框架查询使用 LINQ，因此扩展方法是在 `IQueryable` 和 `IEnumerable` 上定义的。但是，因为它们仅设计为与实体框架一起使用，所以如果尝试在不是实体框架查询的 LINQ 查询中使用它们，则可能会收到以下错误：

源 `IQueryable` 不实现 `IDbAsyncEnumerable<T>`。仅实现 `IDbAsyncEnumerable` 的源可用于实体框架异步操作。
有关详细信息，请参阅<http://go.microsoft.com/fwlink/?LinkId=287068>。

尽管异步方法仅在针对 EF 查询运行时受支持，但在对 `DbSet` 的内存中测试双精度运行时，您可能需要在单元测试中使用它们。

若要使用异步方法，我们需要创建内存中的 `DbAsyncQueryProvider` 以处理异步查询。尽管可以使用 Moq 设置查询提供程序，但使用代码创建测试双实现要容易得多。此实现的代码如下所示：

```

using System.Collections.Generic;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Linq.Expressions;
using System.Threading;

```

```
using System.Threading.Tasks;

namespace TestingDemo
{
    internal class TestDbAsyncQueryProvider<TEntity> : IDbAsyncQueryProvider
    {
        private readonly IQueryProvider _inner;

        internal TestDbAsyncQueryProvider(IQueryProvider inner)
        {
            _inner = inner;
        }

        public IQueryable CreateQuery(Expression expression)
        {
            return new TestDbAsyncEnumerable<TEntity>(expression);
        }

        public IQueryable<TElement> CreateQuery<TElement>(Expression expression)
        {
            return new TestDbAsyncEnumerable<TElement>(expression);
        }

        public object Execute(Expression expression)
        {
            return _inner.Execute(expression);
        }

        public TResult Execute<TResult>(Expression expression)
        {
            return _inner.Execute<TResult>(expression);
        }

        public Task<object> ExecuteAsync(Expression expression, CancellationToken cancellationToken)
        {
            return Task.FromResult(Execute(expression));
        }

        public Task<TResult> ExecuteAsync<TResult>(Expression expression, CancellationToken cancellationToken)
        {
            return Task.FromResult(Execute<TResult>(expression));
        }
    }

    internal class TestDbAsyncEnumerable<T> : EnumerableQuery<T>, IDbAsyncEnumerable<T>, IQueryable<T>
    {
        public TestDbAsyncEnumerable(IEnumerable<T> enumerable)
            : base(enumerable)
        { }

        public TestDbAsyncEnumerable(Expression expression)
            : base(expression)
        { }

        public IDbAsyncEnumerator<T> GetAsyncEnumerator()
        {
            return new TestDbAsyncEnumerator<T>(this.AsEnumerable().GetEnumerator());
        }

        IDbAsyncEnumerator IDbAsyncEnumerable.GetAsyncEnumerator()
        {
            return GetAsyncEnumerator();
        }

        IQueryProvider IQueryable.Provider
        {
            get { return new TestDbAsyncQueryProvider<T>(this); }
        }
    }
}
```

```
internal class TestDbAsyncEnumerator<T> : IDbAsyncEnumerator<T>
{
    private readonly IEnumerator<T> _inner;

    public TestDbAsyncEnumerator(IEnumerator<T> inner)
    {
        _inner = inner;
    }

    public void Dispose()
    {
        _inner.Dispose();
    }

    public Task<bool> MoveNextAsync(CancellationToken cancellationToken)
    {
        return Task.FromResult(_inner.MoveNext());
    }

    public T Current
    {
        get { return _inner.Current; }
    }

    object IDbAsyncEnumerator.Current
    {
        get { return Current; }
    }
}
```

现在，我们有了一个异步查询提供程序，我们可以为新的 GetAllBlogsAsync 方法编写单元测试。

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    [TestClass]
    public class AsyncQueryTests
    {
        [TestMethod]
        public async Task GetAllBlogsAsync_orders_by_name()
        {

            var data = new List<Blog>
            {
                new Blog { Name = "BBB" },
                new Blog { Name = "ZZZ" },
                new Blog { Name = "AAA" },
            }.AsQueryable();

            var mockSet = new Mock<DbSet<Blog>>();
            mockSet.As< IDbAsyncEnumerable<Blog>>()
                .Setup(m => m.GetAsyncEnumerator())
                .Returns(new TestDbAsyncEnumerator<Blog>(data.GetEnumerator()));

            mockSet.As< IQueryable<Blog>>()
                .Setup(m => m.Provider)
                .Returns(new TestDbAsyncQueryProvider<Blog>(data.Provider));

            mockSet.As< IQueryable<Blog>>().Setup(m => m.Expression).Returns(data.Expression);
            mockSet.As< IQueryable<Blog>>().Setup(m => m.ElementType).Returns(data.ElementType);
            mockSet.As< IQueryable<Blog>>().Setup(m => m.GetEnumerator()).Returns(data.GetEnumerator());

            var mockContext = new Mock<BloggingContext>();
            mockContext.Setup(c => c.Blogs).Returns(mockSet.Object);

            var service = new BlogService(mockContext.Object);
            var blogs = await service.GetAllBlogsAsync();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

用自己的测试进行测试双精度

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

编写应用程序测试时，通常需要避免数据库的命中。实体框架使你可以通过创建上下文来实现此目的，该上下文具有测试定义的行为，从而利用内存中数据。

用于创建测试双精度的选项

可使用两种不同的方法创建上下文的内存中版本。

- **创建自己的测试双精度型**-此方法涉及编写您自己的上下文和 dbset 的内存中实现。这使你可以很好地控制类的行为，但可能涉及到编写和拥有合理的代码量。
- **使用模拟框架创建测试双精度型**-使用模拟框架(如 Moq)，可以在运行时动态创建上下文和集。

本文将介绍如何创建自己的测试 double。有关使用模拟框架的信息，请参阅[使用模拟框架进行测试](#)。

用预 EF6 版本测试

本文中所示的代码与 EF6 兼容。若要使用 EF5 和更早的版本进行测试，请参阅[使用虚设上下文进行测试](#)。

EF 内存中测试的限制双精度

内存中测试双精度型可以是一种提供使用 EF 的应用程序的单元测试级别的好方法。但是，在执行此操作时，将使用 LINQ to Objects 对内存中数据执行查询。这可能会导致不同的行为，而不是使用 EF 的 LINQ 提供程序(LINQ to Entities)将查询转换为对数据库运行的 SQL。

这种差异的一个示例就是加载相关数据。如果创建一系列博客，其中每个博客都有相关的文章，则在使用内存中数据时，将始终为每个博客加载相关文章。但是，在对数据库运行时，仅当使用 Include 方法时才会加载数据。

出于此原因，建议始终包括某个级别的端到端测试(除了单元测试)，以确保应用程序能够对数据库正常运行。

与本文一起介绍

本文提供了完整的代码清单，你可以将其复制到 Visual Studio 中，以根据你的需要进行。最简单的方法是创建一个单元测试项目，并需要将 .NET Framework 4.5 作为目标来完成使用 async 的部分。

创建上下文接口

我们将介绍如何测试使用 EF 模型的服务。为了能够将 EF 上下文替换为内存中用于测试的版本，我们将定义 EF 上下文(和内存中双精度)将实现的接口。

我们将要测试的服务将使用上下文的 DbSet 属性来查询和修改数据，还会调用 SaveChanges 将更改推送到数据库。我们将这些成员包括在接口上。

```

using System.Data.Entity;

namespace TestingDemo
{
    public interface IBloggingContext
    {
        DbSet<Blog> Blogs { get; }
        DbSet<Post> Posts { get; }
        int SaveChanges();
    }
}

```

EF 模型

要测试的服务利用的是由 "Bloggingcontext" 和博客和 Post 类组成的 EF 模型。此代码可能是由 EF 设计器生成的，或是 Code First 模型中。

```

using System.Collections.Generic;
using System.Data.Entity;

namespace TestingDemo
{
    public class BloggingContext : DbContext, IBloggingContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }
}

```

通过 EF 设计器实现上下文接口

请注意，我们的上下文实现了 IBloggingContext 接口。

如果使用 Code First，则可以直接编辑上下文来实现接口。如果使用的是 EF 设计器，则需要编辑生成上下文的 T4 模板。打开 > <model_name>.Context.tt 文件嵌套在 edmx 文件下，查找以下代码片段，并将其添加到接口中，如下所示。

```
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> : DbContext, IBloggingContext
```

要测试的服务

为了演示使用内存中测试的测试，我们将为 BlogService 编写一些测试。该服务可以创建新的博客(AddBlog)并返回按名称排序的所有博客(GetAllBlogs)。除了 GetAllBlogs 之外，我们还提供了一个方法，该方法将异步获取按名称(GetAllBlogsAsync)排序的所有博客。

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class BlogService
    {
        private IBloggingContext _context;

        public BlogService(IBloggingContext context)
        {
            _context = context;
        }

        public Blog AddBlog(string name, string url)
        {
            var blog = new Blog { Name = name, Url = url };
            _context.Blogs.Add(blog);
            _context.SaveChanges();

            return blog;
        }

        public List<Blog> GetAllBlogs()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return query.ToList();
        }

        public async Task<List<Blog>> GetAllBlogsAsync()
        {
            var query = from b in _context.Blogs
                        orderby b.Name
                        select b;

            return await query.ToListAsync();
        }
    }
}
```

创建内存中测试双精度

现在，我们已经有了真正的 EF 模型和可使用该模型的服务，接下来可以创建可用于测试的内存中测试双精度。我们为上下文创建了 TestContext 测试 double。在测试中，我们会选择所需的行为，以便支持我们将要运行的测试。在此示例中，我们只是捕获调用 SaveChanges 的次数，但你可以包括验证所测试方案所需的任何逻辑。

我们还创建了 TestDbSet 来提供 DbSet 的内存中实现。我们为 DbSet 上的所有方法提供了一个完整的实现(Find 除外)，但你只需实现测试方案将使用的成员。

TestDbSet 利用我们提供的其他一些基础结构类，以确保可以处理异步查询。

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
```

```

using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Linq.Expressions;
using System.Threading;
using System.Threading.Tasks;

namespace TestingDemo
{
    public class TestContext : IBloggingContext
    {
        public TestContext()
        {
            this.Blogs = new TestDbSet<Blog>();
            this.Posts = new TestDbSet<Post>();
        }

        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
        public int SaveChangesCount { get; private set; }
        public int SaveChanges()
        {
            this.SaveChangesCount++;
            return 1;
        }
    }

    public class TestDbSet<TEntity> : DbSet<TEntity>, IQueryables, Ienumerable<TEntity>, IDbAsyncEnumerable<TEntity>
        where TEntity : class
    {
        ObservableCollection<TEntity> _data;
        IQueryable _query;

        public TestDbSet()
        {
            _data = new ObservableCollection<TEntity>();
            _query = _data.AsQueryable();
        }

        public override TEntity Add(TEntity item)
        {
            _data.Add(item);
            return item;
        }

        public override TEntity Remove(TEntity item)
        {
            _data.Remove(item);
            return item;
        }

        public override TEntity Attach(TEntity item)
        {
            _data.Add(item);
            return item;
        }

        public override TEntity Create()
        {
            return Activator.CreateInstance<TEntity>();
        }

        public override TDerivedEntity Create<TDerivedEntity>()
        {
            return Activator.CreateInstance<TDerivedEntity>();
        }

        public override ObservableCollection<TEntity> Local
    }
}

```

```
{  
    get { return _data; }  
}  
  
Type IQueryable.ElementType  
{  
    get { return _query.ElementType; }  
}  
  
Expression IQueryable.Expression  
{  
    get { return _query.Expression; }  
}  
  
IQueryProvider IQueryable.Provider  
{  
    get { return new TestDbAsyncQueryProvider< TEntity>(_query.Provider); }  
}  
  
System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()  
{  
    return _data.GetEnumerator();  
}  
  
IEnumerator< TEntity> IEnumerable< TEntity>.GetEnumerator()  
{  
    return _data.GetEnumerator();  
}  
  
IDbAsyncEnumerator< TEntity> IDbAsyncEnumerable< TEntity>.GetAsyncEnumerator()  
{  
    return new TestDbAsyncEnumerator< TEntity>(_data.GetEnumerator());  
}  
}  
  
internal class TestDbAsyncQueryProvider< TEntity> : IDbAsyncQueryProvider  
{  
    private readonly IQueryProvider _inner;  
  
    internal TestDbAsyncQueryProvider(IQueryProvider inner)  
    {  
        _inner = inner;  
    }  
  
    public IQueryable CreateQuery(Expression expression)  
    {  
        return new TestDbAsyncEnumerable< TEntity>(expression);  
    }  
  
    public IQueryable< TElement> CreateQuery< TElement>(Expression expression)  
    {  
        return new TestDbAsyncEnumerable< TElement>(expression);  
    }  
  
    public object Execute(Expression expression)  
    {  
        return _inner.Execute(expression);  
    }  
  
    public TResult Execute< TResult>(Expression expression)  
    {  
        return _inner.Execute< TResult>(expression);  
    }  
  
    public Task< object> ExecuteAsync(Expression expression, CancellationToken cancellationToken)  
    {  
        return Task.FromResult(Execute(expression));  
    }  
}
```

```

        public Task<TResult> ExecuteAsync<TResult>(Expression expression, CancellationToken cancellationToken)
    {
        return Task.FromResult(Execute<TResult>(expression));
    }
}

internal class TestDbAsyncEnumerable<T> : EnumerableQuery<T>, IDbAsyncEnumerable<T>, IQueryable<T>
{
    public TestDbAsyncEnumerable(IEnumerable<T> enumerable)
        : base(enumerable)
    { }

    public TestDbAsyncEnumerable(Expression expression)
        : base(expression)
    { }

    public IDbAsyncEnumerator<T> GetAsyncEnumerator()
    {
        return new TestDbAsyncEnumerator<T>(this.AsEnumerable().GetEnumerator());
    }

    IDbAsyncEnumerator IDbAsyncEnumerable.GetAsyncEnumerator()
    {
        return GetAsyncEnumerator();
    }

    IQueryProvider IQueryable.Provider
    {
        get { return new TestDbAsyncQueryProvider<T>(this); }
    }
}

internal class TestDbAsyncEnumerator<T> : IDbAsyncEnumerator<T>
{
    private readonly IEnumerator<T> _inner;

    public TestDbAsyncEnumerator(IEnumerator<T> inner)
    {
        _inner = inner;
    }

    public void Dispose()
    {
        _inner.Dispose();
    }

    public Task<bool> MoveNextAsync(CancellationToken cancellationToken)
    {
        return Task.FromResult(_inner.MoveNext());
    }

    public T Current
    {
        get { return _inner.Current; }
    }

    object IDbAsyncEnumerator.Current
    {
        get { return Current; }
    }
}
}

```

实现查找

Find 方法难以以一般方式实现。如果需要测试使用 Find 方法的代码，最简单的方法是为需要支持 find 的每个实体类型创建测试 DbSet。然后，可以编写逻辑来查找特定类型的实体，如下所示。

```

using System.Linq;

namespace TestingDemo
{
    class TestBlogDbSet : TestDbSet<Blog>
    {
        public override Blog Find(params object[] keyValues)
        {
            var id = (int)keyValues.Single();
            return this.SingleOrDefault(b => b.BlogId == id);
        }
    }
}

```

编写一些测试

这就是开始测试时需要执行的所有操作。以下测试将创建 `TestContext`, 然后基于此上下文创建一个服务。然后, 使用 `AddBlog` 方法创建新的博客。最后, 此测试将验证服务是否已将新的博客添加到上下文的 "博客" 属性, 并在上下文中调用 "SaveChanges"。

这只是您可以使用内存中测试双精度测试的类型的示例, 您可以根据您的要求调整测试的逻辑加倍和验证。

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestingDemo
{
    [TestClass]
    public class NonQueryTests
    {
        [TestMethod]
        public void CreateBlog_saves_a_blog_via_context()
        {
            var context = new TestContext();

            var service = new BlogService(context);
            service.AddBlog("ADO.NET Blog", "http://blogs.msdn.com/adonet");

            Assert.AreEqual(1, context.Blogs.Count());
            Assert.AreEqual("ADO.NET Blog", context.Blogs.Single().Name);
            Assert.AreEqual("http://blogs.msdn.com/adonet", context.Blogs.Single().Url);
            Assert.AreEqual(1, context.SaveChangesCount());
        }
    }
}

```

下面是测试的另一个示例-这是执行查询的另一个示例。首先, 通过在其 "博客" 属性中创建包含某些数据的测试上下文来开始测试, 请注意, 数据不按字母顺序排序。然后, 可以基于测试上下文创建 `BlogService`, 并确保从 `GetAllBlogs` 返回的数据按名称排序。

```

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestingDemo
{
    [TestClass]
    public class QueryTests
    {
        [TestMethod]
        public void GetAllBlogs_orders_by_name()
        {
            var context = new TestContext();
            context.Blogs.Add(new Blog { Name = "BBB" });
            context.Blogs.Add(new Blog { Name = "ZZZ" });
            context.Blogs.Add(new Blog { Name = "AAA" });

            var service = new BlogService(context);
            var blogs = service.GetAllBlogs();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

最后，我们将编写一个使用异步方法的测试，以确保[TestDbSet](#)中包含的异步基础结构工作正常。

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace TestingDemo
{
    [TestClass]
    public class AsyncQueryTests
    {
        [TestMethod]
        public async Task GetAllBlogsAsync_orders_by_name()
        {
            var context = new TestContext();
            context.Blogs.Add(new Blog { Name = "BBB" });
            context.Blogs.Add(new Blog { Name = "ZZZ" });
            context.Blogs.Add(new Blog { Name = "AAA" });

            var service = new BlogService(context);
            var blogs = await service.GetAllBlogsAsync();

            Assert.AreEqual(3, blogs.Count);
            Assert.AreEqual("AAA", blogs[0].Name);
            Assert.AreEqual("BBB", blogs[1].Name);
            Assert.AreEqual("ZZZ", blogs[2].Name);
        }
    }
}

```

可测试性和实体框架4。0

2020/3/11 •

Scott Allen

发布日期:5月2010

介绍

本白皮书介绍并演示了如何通过 ADO.NET 实体框架4.0 和 Visual Studio 2010 编写可测试代码。本文不会尝试专注于特定的测试方法，如测试驱动设计(TDD)或行为驱动设计(BDD)。本文将重点介绍如何编写使用 ADO.NET 实体框架的代码，但仍可以轻松地进行隔离和测试。我们将探讨有助于在数据访问方案中进行测试的常见设计模式，并了解在使用 framework 时如何应用这些模式。我们还将探讨框架的特定功能，以了解这些功能在可测试代码中的工作方式。

什么是可测试代码？

使用自动单元测试来验证软件的功能提供了许多理想的好处。大家都知道，良好的测试会减少应用程序中的软件缺陷数量并提高应用程序的质量，但目前没有找到 bug。

优秀的单元测试套件使开发团队能够节省时间，并仍可控制其创建的软件。团队可以对现有代码进行更改、重构、重新设计和重构软件以满足新要求，并将新组件添加到应用程序中，同时知道测试套件可以验证应用程序的行为。单元测试是快速反馈周期的一部分，用于在复杂性增加的情况下方便更改和保留软件的可维护性。

不过，单元测试有价格。团队必须投入时间来创建和维护单元测试。创建这些测试所需的工作量直接与底层软件的可测试性相关。要测试的软件有多容易？设计具有可测试性的软件的团队将比使用无测试软件的团队更快地创建有效的测试。

Microsoft 设计了 ADO.NET 实体框架4.0 (EF4)，并考虑了可测试性。这并不意味着开发人员将针对框架代码本身编写单元测试。相反，EF4 的可测试性目标使创建在框架之上构建的可测试代码变得简单。在查看具体的示例之前，有必要了解可测试代码的质量。

可测试代码的质量

易于测试的代码将始终至少显示两个特性。首先，可测试代码易于理解。给定一组输入，应该很容易观察代码的输出。例如，测试以下方法非常简单，因为方法直接返回计算的结果。

```
public int Add(int x, int y) {
    return x + y;
}
```

如果方法将计算的值写入网络套接字、数据库表或文件(如以下代码)，则很难测试方法。测试必须执行其他工作来检索值。

```
public void AddAndSaveToFile(int x, int y) {
    var results = string.Format("The answer is {0}", x + y);
    File.WriteAllText("results.txt", results);
}
```

其次，可测试代码易于隔离。让我们使用以下伪代码作为可测试代码的错误示例。

```

public int ComputePolicyValue(InsurancePolicy policy) {
    using (var connection = new SqlConnection("dbConnection"))
    using (var command = new SqlCommand(query, connection)) {
        // business calculations omitted ...

        if (totalValue > notificationThreshold) {
            var message = new MailMessage();
            message.Subject = "Warning!";
            var client = new SmtpClient();
            client.Send(message);
        }
    }
    return totalValue;
}

```

此方法很容易观察—我们可以传入保险政策，并验证返回值是否与预期结果匹配。但是，若要测试方法，我们需要使用正确的架构安装一个数据库，并在该方法尝试发送电子邮件时配置 SMTP 服务器。

单元测试只需验证方法中的计算逻辑，但测试可能会失败，因为电子邮件服务器处于脱机状态，或数据库服务器已移动。这两个故障与测试要验证的行为无关。此行为难以隔离。

努力编写可测试代码的软件开发人员通常会在其编写的代码中保持关注点的分离。上述方法应侧重于业务计算，并将数据库和电子邮件实现细节委托给其他组件。Robert 将按单一责任原则调用此项。对象应封装单个、窄的责任，如计算策略的值。所有其他数据库和通知工作都应负责其他某个对象。以这种方式编写的代码更易于隔离，因为它侧重于单个任务。

在 .NET 中，我们有了需要遵循单一责任原则并实现隔离的抽象。我们可以使用接口定义，并强制代码使用接口抽象，而不是具体类型。在本文的后面部分，我们将了解如何使用类似于上面所示的错误示例的方法来处理与它们将与数据库进行通信的接口。但是在测试时，我们可以用不与数据库通信的虚拟实现替代，而是将数据保存在内存中。此虚拟实现将代码与数据访问代码或数据库配置中不相关的问题隔离开来。

隔离还有其他优点。最后一种方法中的业务计算应该只需几毫秒时间，但测试本身可能会运行几秒钟，作为网络的代码跃点，并与各种服务器通信。单元测试应快速运行以简化小的更改。单元测试也应该是可重复的且不会失败，因为与测试无关的组件有问题。编写易于观察和隔离的代码意味着开发人员可以更轻松地编写代码测试，花费更少的时间等待测试执行，更重要的是，花费更少的时间来跟踪不存在的错误。

希望您可以感激测试的好处，并了解可测试代码展示的质量。我们将介绍如何编写适用于 EF4 的代码，以便将数据保存到数据库中，并将数据保存到数据库中，并使其易于隔离，但首先我们将重点放在讨论数据访问的可测试设计上。

数据持久性的设计模式

前面介绍的两个错误示例都具有太多的责任。第一个错误示例必须对文件执行计算和写入。第二个错误示例是从数据库中读取数据并执行业务计算并发送电子邮件。通过设计分隔关注点并将责任委派给其他组件的较小方法，你将在编写可测试代码时做出很大努力。其目标是通过编写小型和重点抽象的操作来生成功能。

当涉及数据持久性时，我们所要寻找的小型重点抽象非常常见。圣马丁 Fowler 企业应用程序体系结构的书籍模式是第一次在打印中描述这些模式。在以下部分中，我们将简要介绍这些模式，然后显示这些 ADO.NET 实体框架如何实现和使用这些模式。

存储库模式

Fowler 显示了一个存储库“使用类似于集合的接口来访问域对象”的域和数据映射层。存储库模式的目标是将代码与数据访问的 minutiae 隔离，因此，我们看到的早期隔离是可测试性的必需特性。

隔离的关键是存储库如何使用类似集合的接口公开对象。你编写的使用存储库的逻辑并不知道存储库将如何具体化你请求的对象。存储库可能会与数据库进行通信，也可能只是从内存中集合返回对象。您的所有代码都需要知道存储库似乎维护集合，并且您可以从集合中检索、添加和删除对象。

在现有的 .NET 应用程序中，具体的存储库通常继承自如下所示的泛型接口：

```
public interface IRepository<T> {
    IEnumerable<T> FindAll();
    IEnumerable<T> FindBy(Expression<Func<T, bool>> predicate);
    T FindById(int id);
    void Add(T newEntity);
    void Remove(T entity);
}
```

当我们为 EF4 提供实现时，我们将对接口定义进行一些更改，但基本概念仍保持不变。代码可以使用实现此接口的具体存储库，根据谓词的计算结果检索实体集合，或者只检索所有可用的实体。代码还可以通过存储库接口添加和删除实体。

对于员工对象的 IRepository，代码可以执行以下操作。

```
var employeesNamedScott =
    repository
        .FindBy(e => e.Name == "Scott")
        .OrderBy(e => e.HireDate);
var firstEmployee = repository.FindById(1);
var newEmployee = new Employee() { /*... */};
repository.Add(newEmployee);
```

由于代码使用的是接口 (IRepository)，因此我们可以为代码提供不同的接口实现。一个实现可能是 EF4 支持的实现，并将对象保存到 Microsoft SQL Server 数据库中。不同的实现（我们在测试过程中使用的）可能由一个内存中员工对象列表来支持。接口将帮助实现代码中的隔离。

请注意， IRepository<T> 接口不会公开保存操作。如何更新现有对象？你可能会遇到包含 Save 操作的 IRepository 定义，而这些存储库的实现需要立即将对象保存到数据库中。但是，在许多应用程序中，我们不希望单独保存对象。相反，我们想要将对象从不同的存储库中移到现实中，并将这些对象修改为业务活动的一部分，然后将所有对象作为单个原子操作的一部分进行保存。幸运的是，有一种模式可用于实现此类型的行为。

工作单元模式

Fowler 表示工作单元将“维护受业务事务影响的对象列表，并协调发生的更改和并发问题的解决方法”。工作单元的责任是跟踪对存储库所做的对象所做的更改，并在告知工作单元提交更改时保留对对象所做的任何更改。还需要负责将已添加的新对象添加到所有存储库中的工作单元，并将对象插入到数据库中，还会管理删除。

如果曾经使用过 ADO.NET 数据集完成任何工作，则已熟悉工作单元模式。ADO.NET 数据集能够跟踪对 DataRow 对象的更新、删除和插入操作，并且可以（使用 TableAdapter 的帮助）协调对数据库进行的所有更改。但是， DataSet 对象会为底层数据库的断开连接的子集建模。工作单元模式的行为相同，但它适用于与数据访问代码隔离并不知道数据库的业务对象和域对象。

对 .NET 代码中的工作单元进行建模的抽象可能如下所示：

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<Order> Orders { get; }
    IRepository<Customer> Customers { get; }
    void Commit();
}
```

通过从工作单元公开存储库引用，可以确保单个工作单元对象能够跟踪在业务事务中具体化的所有实体。在实际工作中实现 Commit 方法是指所有神奇的情况都是为了协调与数据库的内存中更改。

给定 IUnitOfWork 引用后，代码可以对从一个或多个存储库检索的业务对象进行更改，并使用原子提交操作保存所有更改。

```
var firstEmployee = unitofWork.Employees.FindById(1);
var firstCustomer = unitofWork.Customers.FindById(1);
firstEmployee.Name = "Alex";
firstCustomer.Name = "Christopher";
unitofWork.Commit();
```

延迟负载模式

Fowler 使用名称**懒惰加载**来描述 "对象，该对象不包含你需要的所有数据，但知道如何获取它"。在编写可测试的业务代码和使用关系数据库时，透明延迟加载是一项重要功能。作为示例，请考虑以下代码。

```
var employee = repository.FindById(id);
// ... and later ...
foreach(var timeCard in employee.TimeCards) {
    // .. manipulate the timeCard
}
```

如何填充时间卡片集合？有两个可能的答案。一项答案是，如果员工需要提取员工，则会发出一个查询，同时检索员工和员工的关联的时间卡信息。在关系数据库中，这通常需要带有 JOIN 子句的查询，并且可能会导致检索超过应用程序需求的信息。如果应用程序永远不需要接触时间卡片属性，该怎么办？

第二个答案是按需加载时间卡片属性。此延迟加载对于业务逻辑是隐式且透明的，因为代码不会调用特殊的 API 来检索时间卡信息。此代码假定需要时提供卡信息。延迟加载涉及一些神奇的延迟，通常涉及方法调用的运行时侦听。拦截代码负责与数据库进行通信并检索时间卡信息，同时使业务逻辑成为业务逻辑。这种延迟加载幻数允许业务代码将自身与数据检索操作隔离开来，并导致代码更具可测试性。

延迟加载的缺点是，当应用程序需要时间卡信息时，代码将执行其他查询。对于许多应用程序而言，这并不是问题，但对于性能敏感的应用程序或应用程序在循环的每次迭代期间循环使用查询来检索时间卡（通常称为 N + 1 查询问题），延迟加载是一个拖动。在这些情况下，应用程序可能需要以最有效的方式积极加载时间卡信息。

幸运的是，在转到下一节并实现这些模式时，我们将了解 EF4 如何支持隐式延迟加载和高效的预先加载。

用实体框架实现模式

好消息是，我们在上一节中介绍的所有设计模式都是通过 EF4 实现的简单方式。为了演示，我们将使用一个简单的 ASP.NET MVC 应用程序来编辑和显示员工及其相关的时间卡信息。首先，我们将使用以下 "纯旧 CLR 对象 (Poco)"。

```
public class Employee {
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime HireDate { get; set; }
    public ICollection<TimeCard> TimeCards { get; set; }
}

public class TimeCard {
    public int Id { get; set; }
    public int Hours { get; set; }
    public DateTime EffectiveDate { get; set; }
}
```

当我们探索 EF4 的不同方法和功能时，这些类定义将略有不同，但其目的是将这些类视为持久性未知 (PI)。PI 对象并不知道它持有的状态在数据库内的状态。PI 和 Poco 有可测试软件。使用 POCO 方法的对象不受约束，更灵活，更容易测试，因为它们可以在没有数据库的情况下运行。

通过 Poco，我们可以在 Visual Studio 中创建实体数据模型 (EDM)（参见图1）。我们不会使用 EDM 为实体生成代码。相反，我们想要使用我们 lovingly 的实体。我们将仅使用 EDM 生成数据库架构，并提供元数据 EF4 需要将对象映射到数据库。

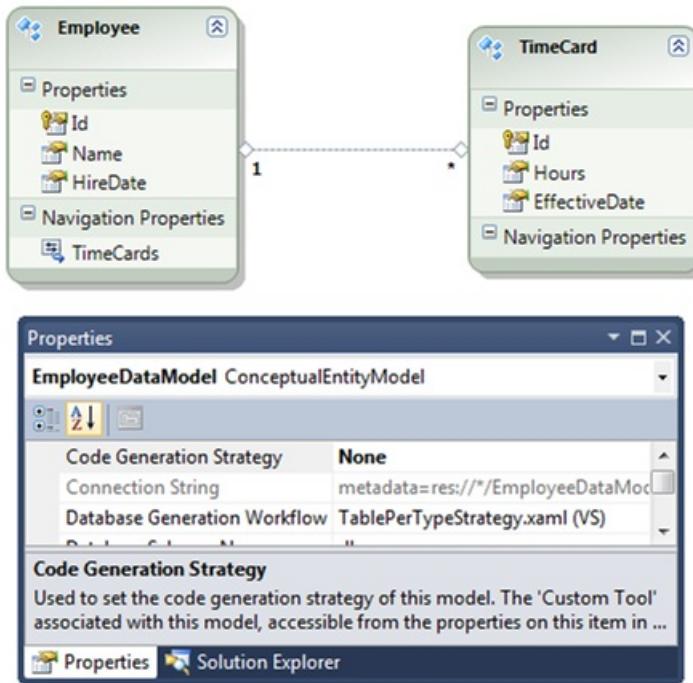


图1

注意：如果要首先开发 EDM 模型，可以从 EDM 生成干净的 POCO 代码。你可以使用数据可编程性团队提供的 Visual Studio 2010 扩展来执行此操作。若要下载该扩展，请从 Visual Studio 中的“工具”菜单启动“扩展管理器”，然后在“POCO”模板的联机库中搜索（参见图2）。有多种可用于 EF 的 POCO 模板。有关使用模板的详细信息，请参阅“[演练：用于实体框架的 POCO 模板](#)”。

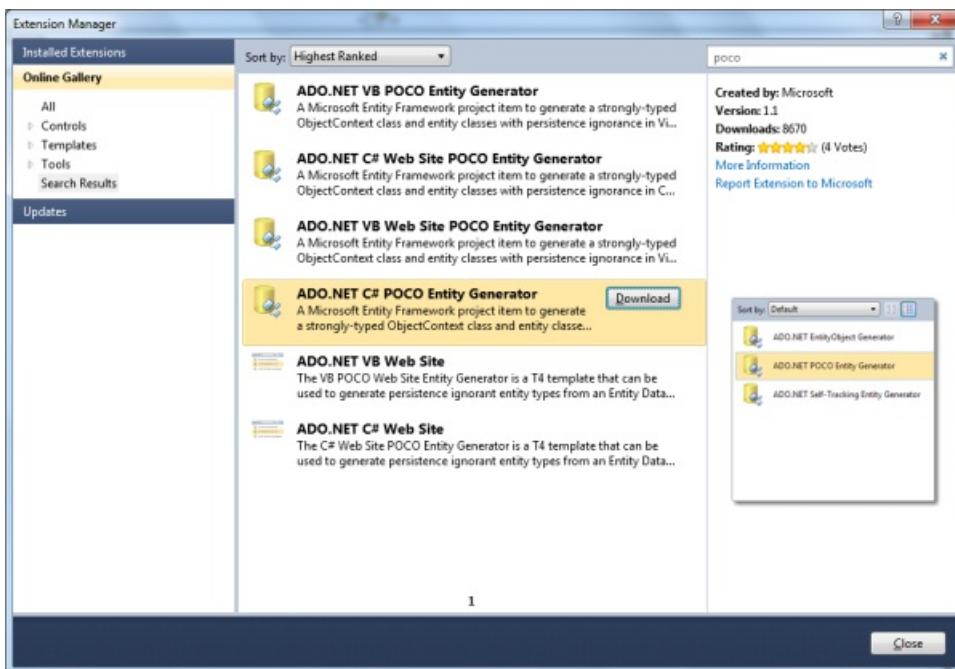


图2

从此 POCO 开始，我们将探讨可测试代码的两种不同方法。第一种方法是调用 EF 方法，因为它利用实体框架 API 中的抽象来实现工作单元和存储库。在第二种方法中，我们将创建自己的自定义存储库抽象，并了解每种方法的优点和缺点。首先，我们将探讨 EF 方法。

EF 中心实现

请考虑 ASP.NET MVC 项目中的以下控制器操作。操作将检索 Employee 对象，并返回一个结果以显示雇员的详细视图。

```
public ViewResult Details(int id) {
    var employee = _unitOfWork.Employees
        .Single(e => e.Id == id);
    return View(employee);
}
```

代码是否可测试？至少需要两个测试来验证操作的行为。首先，我们想要验证操作是否返回正确的视图—一个简单的测试。我们还希望编写一个测试来验证该操作是否可检索正确的雇员，我们想要执行此操作，而不执行代码查询数据库。请记住，我们想要隔离受测代码。隔离将确保测试不会因数据访问代码或数据库配置中的错误而失败。如果测试失败，我们将了解控制器逻辑中的 bug，而不是在某个较低级别的系统组件中。

若要实现隔离，我们需要一些抽象，如我们之前为存储库和工作单元提供的接口。请记住，存储库模式旨在协调域对象和数据映射层。在此方案中，EF4 是数据映射层，并且已经提供名为 `IObjectSet` 的存储库抽象抽象`<T>`（来自 `System.Object` 命名空间）。接口定义如下所示。

```
public interface IObjectSet< TEntity > :
    IQueryable< TEntity >,
    IEnumerable< TEntity >,
    IQueryable,
    IEnumerable
    where TEntity : class
{
    void AddObject(TEntity entity);
    void Attach(TEntity entity);
    void DeleteObject(TEntity entity);
    void Detach(TEntity entity);
}
```

`IObjectSet<T>` 满足存储库的要求，因为它类似于对象的集合（通过 `IEnumerable<>`），并提供了在模拟的集合中添加和删除对象的方法。附加和分离方法公开了 EF4 API 的其他功能。若要使用 `IObjectSet<T>` 作为存储库的接口，我们需要一个工作单元来将存储库绑定在一起。

```
public interface IUnitOfWork {
    IObjectSet<Employee> Employees { get; }
    IObjectSet<TimeCard> TimeCards { get; }
    void Commit();
}
```

此接口的一个具体实现将与 SQL Server 通信，并使用 EF4 中的 `ObjectContext` 类轻松创建。`ObjectContext` 类是 EF4 API 中的实际工作单元。

```

public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        var connectionString =
            ConfigurationManager
                .ConnectionStrings[ConnectionStringName]
                .ConnectionString;
        _context = new ObjectContext(connectionString);
    }

    public IObjectSet<Employee> Employees {
        get { return _context.CreateObjectSet<Employee>(); }
    }

    public IObjectSet<TimeCard> TimeCards {
        get { return _context.CreateObjectSet<TimeCard>(); }
    }

    public void Commit() {
        _context.SaveChanges();
    }

    readonly ObjectContext _context;
    const string ConnectionStringName = "EmployeeDataModelContainer";
}

```

将 `IObjectSet<T>` 的生活与调用 `ObjectContext` 对象的 `Createobjectset` 方法一样简单。在后台，框架将使用我们在 EDM 中提供的元数据生成具体的 `ObjectSet<T>`。我们会坚持返回 `IObjectSet<T>` 接口，因为它有助于在客户端代码中保留可测试性。

这种具体的实现在生产中很有用，但我们需要重点介绍我们如何使用我们的 `IUnitOfWork` 抽象来简化测试。

测试双精度

若要隔离控制器操作，我们需要能够在实际工作单元(由 `ObjectContext` 支持)和测试 `double` 或 "假" 工作单元(执行内存中操作)之间进行切换。执行这种类型的切换的常见方法是不让 MVC 控制器实例化工作单元，而是将工作单元作为构造函数参数传递到控制器。

```

class EmployeeController : Controller {
    public EmployeeController(IUnitOfWork unitOfWork) {
        _unitOfWork = unitOfWork;
    }
    ...
}

```

上面的代码是依赖关系注入的示例。我们不允许控制器创建依赖项(工作单元)，但会将依赖项注入到控制器中。在 MVC 项目中，通常将自定义控制器工厂与控制反转(IoC)容器结合使用来自动执行依赖关系注入。这些主题超出了本文的范围，但你可以通过遵循本文末尾的参考来了解详细信息。

可用于测试的伪工作单元实现可能如下所示。

```
public class InMemoryUnitOfWork : IUnitOfWork {
    public InMemoryUnitOfWork() {
        Committed = false;
    }
    public IObjectSet<Employee> Employees {
        get;
        set;
    }

    public IObjectSet<TimeCard> TimeCards {
        get;
        set;
    }

    public bool Committed { get; set; }
    public void Commit() {
        Committed = true;
    }
}
```

请注意，伪工作单元公开了提交的属性。有时将功能添加到可简化测试的虚设类中。在这种情况下，如果代码通过检查确认属性来提交工作单元，则很容易观察。

还需要一个虚设的 `IObjectSet<T>` 将员工和卡上的对象保存在内存中。我们可以使用泛型提供单个实现。

```

public class InMemoryObjectSet<T> : IObjectSet<T> where T : class
{
    public InMemoryObjectSet()
        : this(Enumerable.Empty<T>()) {
    }

    public InMemoryObjectSet(IEnumerable<T> entities) {
        _set = new HashSet<T>();
        foreach (var entity in entities) {
            _set.Add(entity);
        }
        _queryableSet = _set.AsQueryable();
    }

    public void AddObject(T entity) {
        _set.Add(entity);
    }

    public void Attach(T entity) {
        _set.Add(entity);
    }

    public void DeleteObject(T entity) {
        _set.Remove(entity);
    }

    public void Detach(T entity) {
        _set.Remove(entity);
    }

    public Type ElementType {
        get { return _queryableSet.ElementType; }
    }

    public Expression Expression {
        get { return _queryableSet.Expression; }
    }

    public IQueryProvider Provider {
        get { return _queryableSet.Provider; }
    }

    public IEnumerator<T> GetEnumerator() {
        return _set.GetEnumerator();
    }

    Ienumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }

    readonly HashSet<T> _set;
    readonly IQueryable<T> _queryableSet;
}

```

此测试将其大部分工作委托给基础 `HashSet<T>` 对象。请注意，`IObjectSet<T>` 需要将 `T` 强制为类（引用类型）的泛型约束，并强制我们实现 `IQueryable<>`。使用标准 LINQ 运算符 `AsQueryable`，可以轻松地将内存中集合显示为 `IQueryable<T>`。

测试

传统单元测试将使用一个测试类来保存单个 MVC 控制器中所有操作的所有测试。我们可以使用生成的内存 fakes 中的来编写这些测试或任何类型的单元测试。但在本文中，我们将避免使用整体测试类方法，而是将测试分组，以重点关注特定功能。例如，“创建新员工”可能是我们要测试的功能，因此，我们将使用一个测试类来验证负责创建新员工的单个控制器操作。

所有这些细化测试类都需要一些常见的安装代码。例如，我们始终需要创建内存中存储库和伪工作单元。还需要一个员工控制器实例，其中包含虚假的工作单元。我们将使用基类在测试类之间共享此常见安装代码。

```

public class EmployeeControllerTestBase {
    public EmployeeControllerTestBase() {
        _employeeData = EmployeeObjectMother.CreateEmployees()
            .ToList();
        _repository = new InMemoryObjectSet<Employee>(_employeeData);
        _unitOfWork = new InMemoryUnitOfWork();
        _unitOfWork.Employees = _repository;
        _controller = new EmployeeController(_unitOfWork);
    }

    protected IList<Employee> _employeeData;
    protected EmployeeController _controller;
    protected InMemoryObjectSet<Employee> _repository;
    protected InMemoryUnitOfWork _unitOfWork;
}

```

在基类中使用的 "对象母亲" 是创建测试数据的一种常见模式。对象母亲包含用于实例化测试实体的工厂方法，以便在多个测试装置中使用。

```

public static class EmployeeObjectMother {
    public static IEnumerable<Employee> CreateEmployees() {
        yield return new Employee() {
            Id = 1, Name = "Scott", HireDate=new DateTime(2002, 1, 1)
        };
        yield return new Employee() {
            Id = 2, Name = "Poonam", HireDate=new DateTime(2001, 1, 1)
        };
        yield return new Employee() {
            Id = 3, Name = "Simon", HireDate=new DateTime(2008, 1, 1)
        };
    }
    // ... more fake data for different scenarios
}

```

我们可以将 EmployeeControllerTestBase 用作多个测试装置的基类(请参阅图3)。每个测试装置将测试特定控制器操作。例如，一个测试装置将重点放在测试 HTTP GET 请求期间使用的 "创建" 操作(用于显示用于创建员工的视图)，另一个装置将重点放在 HTTP POST 请求中使用的 "创建" 操作(以获取由用户创建员工)。每个派生类只负责特定上下文中所需的设置，并提供验证其特定测试上下文的结果所需的断言。

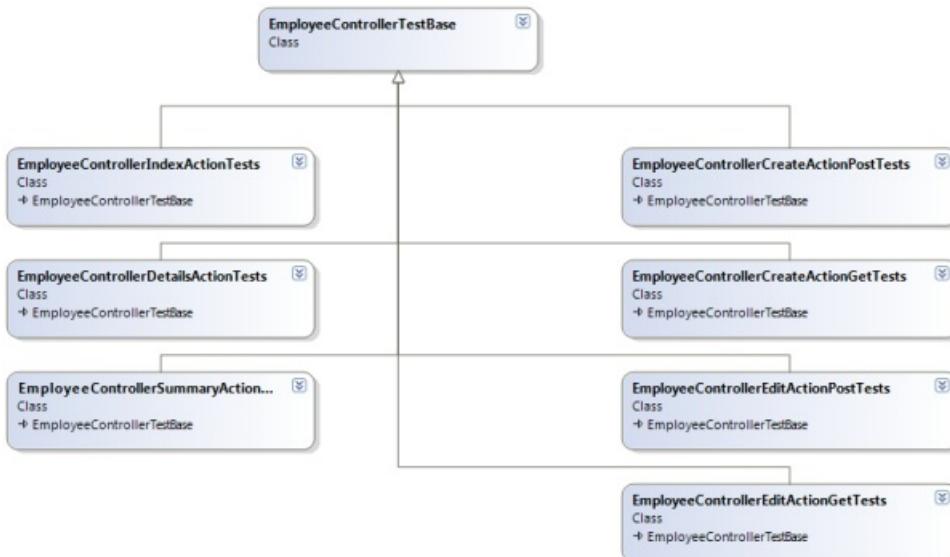


图3

此处提供的命名约定和测试样式不是可测试代码所必需的—它只是一种方法。图4显示了 Visual Studio 2010 的 Jet 大脑 Resharper 测试运行程序插件中运行的测试。

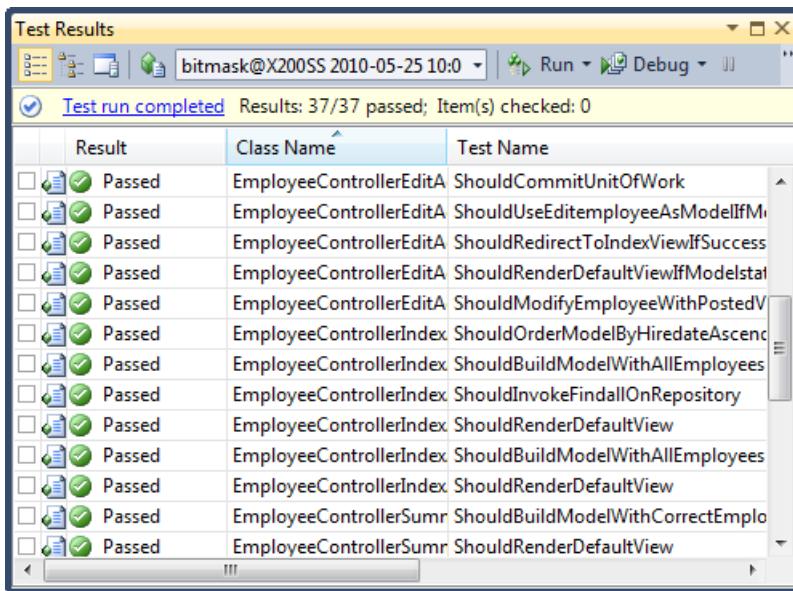


图4

使用基类来处理共享的安装代码，每个控制器操作的单元测试都很小，并且易于编写。测试将快速执行(因为我们在执行内存中的操作)，不应因为不相关的基础结构或环境问题而失败(因为我们已将所测试的单元隔离)。

```
[TestClass]
public class EmployeeControllerCreateActionPostTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldAddNewEmployeeToRepository() {
        _controller.Create(_newEmployee);
        Assert.IsTrue(_repository.Contains(_newEmployee));
    }
    [TestMethod]
    public void ShouldCommitUnitOfWork() {
        _controller.Create(_newEmployee);
        Assert.IsTrue(_unitOfWork.Committed);
    }
    // ... more tests
    Employee _newEmployee = new Employee() {
        Name = "NEW EMPLOYEE",
        HireDate = new System.DateTime(2010, 1, 1)
    };
}
```

在这些测试中，基类执行大部分的设置工作。请记住，基类构造函数会创建内存中存储库、伪工作单元和 EmployeeController 类的实例。该测试类从此基类派生，重点介绍了测试 Create 方法的具体信息。在这种情况下，将在任何单元测试过程中看到的“排列、操作和断言”步骤可归结：

- 创建用于模拟传入数据的 newEmployee 对象。
- 调用 EmployeeController 的 Create 操作并传入 newEmployee。
- 验证“创建”操作是否生成预期的结果(员工出现在存储库中)。

我们生成的内容使我们可以测试任何 EmployeeController 操作。例如，当我们为 Employee 控制器的索引操作编写测试时，可以从测试基类继承，为测试建立相同的基本设置。同样，基类会创建内存中存储库、伪工作单元和 EmployeeController 的实例。针对索引操作的测试只需重点介绍如何调用索引操作并测试操作返回的模型的质量。

```

[TestClass]
public class EmployeeControllerIndexActionTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldBuildModelWithAllEmployees() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.Count() == _employeeData.Count);
    }
    [TestMethod]
    public void ShouldOrderModelByHiredateAscending() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.SequenceEqual(
            _employeeData.OrderBy(e => e.HireDate)));
    }
    // ...
}

```

用内存中 fakes 创建的测试面向测试软件的状态。例如，在测试创建操作时，我们想要在执行 "创建" 操作后检查存储库的状态—存储库是否持有新员工？

```

[TestMethod]
public void ShouldAddNewEmployeeToRepository() {
    _controller.Create(_newEmployee);
    Assert.IsTrue(_repository.Contains(_newEmployee));
}

```

稍后我们将探讨基于交互的测试。基于交互的测试会询问所测试的代码是否在对象上调用了正确的方法，并传递了正确的参数。现在，我们将继续介绍另一种设计模式—延迟加载。

预先加载和延迟加载

在 ASP.NET MVC web 应用程序的某个时候，我们可能希望显示雇员的信息，并包括员工的相关时间表。例如，我们可能会显示一个 "时间表摘要" 显示，其中显示了雇员的姓名和系统中的总时间。可以采用几种方法实现此功能。

投影

创建摘要的一种简单方法是构造专用于要在视图中显示的信息的模型。在这种情况下，该模型可能如下所示。

```

public class EmployeeSummaryViewModel {
    public string Name { get; set; }
    public int TotalTimeCards { get; set; }
}

```

请注意，EmployeeSummaryViewModel 不是一个实体—换言之，在数据库中不会保留它。我们只会使用此类以强类型方式将数据无序播放到视图中。视图模型类似于数据传输对象(DTO)，因为它不包含 "仅行为" 属性。这些属性将保存需要移动的数据。使用 LINQ 的标准投影运算符(Select 运算符)，可以轻松地实例化此视图模型。

```

public ViewResult Summary(int id) {
    var model = _unitOfWork.Employees
        .Where(e => e.Id == id)
        .Select(e => new EmployeeSummaryViewModel
    {
        Name = e.Name,
        TotalTimeCards = e.TimeCards.Count()
    })
    .Single();
    return View(model);
}

```

上面的代码有两个值得注意的功能。首先—代码易于测试，因为它仍易于观察和隔离。除了实际工作单元，Select 运算符还能与内存中的 fakes 一起工作。

```

[TestClass]
public class EmployeeControllerSummaryActionTests
    : EmployeeControllerTestBase {
    [TestMethod]
    public void ShouldBuildModelWithCorrectEmployeeSummary() {
        var id = 1;
        var result = _controller.Summary(id);
        var model = result.ViewData.Model as EmployeeSummaryViewModel;
        Assert.IsTrue(model.TotalTimeCards == 3);
    }
    // ...
}

```

第二个值得注意的功能是代码如何允许 EF4 生成单个有效的查询，以将员工和时间卡信息组合在一起。我们已将员工信息和时间卡信息加载到同一个对象中，而无需使用任何特殊的 API。此代码仅使用标准 LINQ 运算符（适用于内存中数据源以及远程数据源）来表示所需的信息。EF4 能够将 LINQ 查询和 C# 编译器生成的表达式树转换为单个高效的 T-SQL 查询。

```

SELECT
[LIMIT1].[Id] AS [Id],
[LIMIT1].[Name] AS [Name],
[LIMIT1].[C1] AS [C1]
FROM (SELECT TOP (2)
[Project1].[Id] AS [Id],
[Project1].[Name] AS [Name],
[Project1].[C1] AS [C1]
FROM (SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Name] AS [Name],
(SELECT COUNT(1) AS [A1]
FROM [dbo].[TimeCards] AS [Extent2]
WHERE [Extent1].[Id] =
[Extent2].[EmployeeTimeCard_TimeCard_Id]) AS [C1]
FROM [dbo].[Employees] AS [Extent1]
WHERE [Extent1].[Id] = @p_linq_0
) AS [Project1]
) AS [LIMIT1]

```

在其他一些情况下，我们不想使用视图模型或 DTO 对象，而是使用实际实体。当我们知道我们需要员工和员工的时间卡时，可以积极以不引入注目和高效的方式加载相关数据。

显式预先加载

当我们想要积极加载相关实体信息时，我们需要某种机制来实现业务逻辑（或在此方案中为控制器操作逻辑），以表达其对存储库的期望。EF4 ObjectQuery<T> 类定义了包含方法，以指定在查询过程中要检索的相关对象。请记住，EF4 ObjectContext 通过 > 类的具体 ObjectSet<类公开实体>，后者继承自 ObjectQuery<T>。如果使用的是

ObjectSet<T> 控制器操作中的引用，我们可以编写以下代码来指定每个员工的时间卡信息的预先加载。

```
_employees.Include("TimeCards")
    .Where(e => e.HireDate.Year > 2009);
```

不过，由于我们要尝试让代码可测试，因此，我们不会公开 ObjectSet<T> 从真实的工作单元的外部。相反，我们依赖于 IObjectSet<T> 接口，该接口更易于假冒，但 IObjectSet<T> 未定义 Include 方法。LINQ 的优点是，我们可以创建自己的 Include 运算符。

```
public static class QueryableExtensions {
    public static IQueryable<T> Include<T>
        (this IQueryable<T> sequence, string path) {
            var objectQuery = sequence as ObjectQuery<T>;
            if(objectQuery != null)
            {
                return objectQuery.Include(path);
            }
            return sequence;
        }
}
```

请注意，此 Include 运算符定义为 IQueryable<T> 的扩展方法，而不是> 的 IObjectSet<。这使我们能够将方法用于范围更广的可能类型，包括 IQueryable<T>、IObjectSet<T>、ObjectQuery<T> 和 ObjectSet<T>。如果基础序列不是真正的 EF4 ObjectQuery<T>，则不会有任何损害，而且 Include 运算符是一个无操作。如果基础序列是ObjectQuery<t>（或派生自 ObjectQuery<>），则 EF4 将看到对其他数据的要求，并制定正确的 SQL 查询。

使用这个新的运算符，我们可以从存储库显式请求预先加载的时间卡信息。

```
public ViewResult Index() {
    var model = _unitOfWork.Employees
        .Include("TimeCards")
        .OrderBy(e => e.HireDate);
    return View(model);
}
```

针对实际 ObjectContext 运行时，代码将生成以下单个查询。查询在一次中从数据库收集足够多的信息，以具体化 employee 对象并完全填充其时间卡片属性。

```

SELECT
[Project1].[Id] AS [Id],
[Project1].[Name] AS [Name],
[Project1].[HireDate] AS [HireDate],
[Project1].[C1] AS [C1],
[Project1].[Id1] AS [Id1],
[Project1].[Hours] AS [Hours],
[Project1].[EffectiveDate] AS [EffectiveDate],
[Project1].[EmployeeTimeCard_TimeCard_Id] AS [EmployeeTimeCard_TimeCard_Id]
FROM ( SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Name] AS [Name],
[Extent1].[HireDate] AS [HireDate],
[Extent2].[Id] AS [Id1],
[Extent2].[Hours] AS [Hours],
[Extent2].[EffectiveDate] AS [EffectiveDate],
[Extent2].[EmployeeTimeCard_TimeCard_Id] AS
[EmployeeTimeCard_TimeCard_Id],
CASE WHEN ([Extent2].[Id] IS NULL) THEN CAST(NULL AS int)
ELSE 1 END AS [C1]
FROM [dbo].[Employees] AS [Extent1]
LEFT OUTER JOIN [dbo].[TimeCards] AS [Extent2] ON [Extent1].[Id] = [Extent2].
[EmployeeTimeCard_TimeCard_Id]
) AS [Project1]
ORDER BY [Project1].[HireDate] ASC,
[Project1].[Id] ASC, [Project1].[C1] ASC

```

好消息是，操作方法中的代码保持完全可测试性。我们不需要为 fakes 提供任何其他功能来支持 Include 运算符。糟糕的是，我们必须在要保留持久性未知的代码内使用 Include 运算符。这是生成可测试代码时需要评估的折衷类型的一个典型示例。有时，您需要使持久性问题在存储库抽象之外泄露以满足性能目标。

预先加载的替代方法是延迟加载。延迟加载意味着我们不需要业务代码来显式公告关联数据的要求。相反，我们在应用程序中使用实体，如果需要其他数据实体框架将按需加载数据。

延迟加载

这种情况很容易想到，这种情况下，我们并不知道有业务逻辑需要哪些数据。我们可能知道逻辑需要一个 employee 对象，但我们可能会将其分支到不同的执行路径，其中某些路径需要员工的时间卡信息，而另一些则不需要。这种情况非常适合隐式延迟加载，因为数据神奇地出现在需要的基础上。

延迟加载(也称为延迟加载)对实体对象有一些要求。具有 true 持久性无感知的 Poco 不会面对持久性层的任何要求，但真正的持久性无感知根本无法实现。相反，我们会以相对度数度量持久性无感知。如果需要从面向持久性的基类继承，或使用专用集合在 Poco 中实现延迟加载，则会很遗憾。幸运的是，EF4 具有更不具侵入性的解决方案。

几乎无法检测

使用 POCO 对象时，EF4 可以为实体动态生成运行时代理。这些代理以不可见的方式包装具体化的 Poco，并通过截取每个属性 get 和 set 操作来执行其他工作，从而提供其他服务。其中一项服务是我们要查找的延迟加载功能。另一服务是一种有效的更改跟踪机制，可以在程序更改实体的属性值时进行记录。在 SaveChanges 方法期间 ObjectContext 使用更改列表，以使用更新命令来持久保存所有已修改的实体。

但是，若要使这些代理正常工作，它们需要有一种方法来挂钩实体上的属性 get 和 set 操作，并且代理通过重写虚拟成员来实现此目标。因此，如果需要隐式延迟加载和高效的更改跟踪，则需要返回到 POCO 类定义并将属性标记为虚拟。

```

public class Employee {
    public virtual int Id { get; set; }
    public virtual string Name { get; set; }
    public virtual DateTime HireDate { get; set; }
    public virtual ICollection<TimeCard> TimeCards { get; set; }
}

```

我们仍然可以说，Employee 实体主要是持久性未知的。唯一的要求是使用虚拟成员，这不会影响代码的可测试性。不需要从任何特殊基类派生，甚至可以使用专用于延迟加载的特殊集合。如代码所示，任何实现 ICollection<T> 的类都可用于保存相关实体。

还需要在我们的工作单元内进行一个小小的更改。当直接使用 ObjectContext 对象时，延迟加载默认情况下处于关闭状态。可以在 ContextOptions 属性上设置一个属性来启用延迟加载，如果我们想要在任何位置启用延迟加载，则可以在实际工作单元内设置此属性。

```
public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        // ...
        _context = new ObjectContext(connectionString);
        _context.ContextOptions.LazyLoadingEnabled = true;
    }
    // ...
}
```

启用隐式延迟加载后，应用程序代码可以使用员工和员工的相关时间表，同时丝毫不知道 EF 加载额外数据所需的工作。

```
var employee = _unitOfWork.Employees
    .Single(e => e.Id == id);
foreach (var card in employee.TimeCards) {
    // ...
}
```

延迟加载使得应用程序代码更易于编写，而对于代理的神奇，代码会保持完全可测试性。工作单元的内存中 fakes 只需在测试过程中需要时使用关联数据预加载虚设实体。

此时，我们将使用 IObjectSet<T> 来找出生成存储库的注意力，并查看抽象以隐藏持久性框架的所有标志。

自定义存储库

当我们在本文中第一次介绍工作单元设计模式时，我们提供了一些示例代码，说明工作单元的外观。让我们使用我们一直在使用的员工和员工时间表方案来重新展示这一思路。

```
public interface IUnitOfWork {
    IRepository<Employee> Employees { get; }
    IRepository<TimeCard> TimeCards { get; }
    void Commit();
}
```

此工作单元和我们在上一节中创建的工作单元之间的主要区别在于，此工作单元不使用 EF4 框架中的任何抽象(>没有任何 IObjectSet<>)。IObjectSet<T> 可用作存储库接口，但它公开的 API 可能并不完全符合应用程序的需求。在即将推出的方法中，我们将使用自定义 IRepository<T> 抽象来表示存储库。

许多遵循测试驱动设计、行为驱动设计和域驱动方法设计的开发人员出于多种原因而 IRepository<T> 方法。首先， IRepository<T> 接口表示 "反损坏" 层。正如 Eric Evans 在他的域驱动的设计书中所述，抗损坏层会使你的域代码远离基础结构 Api，如永久性 API。其次，开发人员可以在存储库中构建满足应用程序确切需求的方法(在编写测试时发现)。例如，我们可能经常需要使用 ID 值查找单个实体，因此可以将 FindById 方法添加到存储库接口。 IRepository<T> 定义将如下所示。

```

public interface IRepository<T>
    where T : class, IEntity {
    IQueryable<T> FindAll();
    IQueryable<T> FindWhere(Expression<Func<T, bool>> predicate);
    T FindById(int id);
    void Add(T newEntity);
    void Remove(T entity);
}

```

请注意，我们将返回使用 `IQueryable<T>` 接口来公开实体集合。`IQueryable<T>` 允许 LINQ 表达式树流入 EF4 提供程序，并为提供程序提供查询的整体视图。第二个选项是返回 `IEnumerable<T>`，这意味着 EF4 LINQ 提供程序将仅查看在存储库内生成的表达式。不会将存储库外部完成的任何分组、排序和投影组合到发送到数据库的 SQL 命令中，这可能会影响性能。另一方面，只返回 `IEnumerable<T>` 结果的存储库永远不会让你使用新的 SQL 命令。这两种方法都适用，并且这两种方法保持可测试性。

简单的方法是使用泛型和 EF4 `ObjectContext` API 提供 `IRepository<T>` 接口的单个实现。

```

public class SqlRepository<T> : IRepository<T>
    where T : class, IEntity {
    public SqlRepository(ObjectContext context) {
        _objectSet = context.CreateObjectSet<T>();
    }
    public IQueryable<T> FindAll() {
        return _objectSet;
    }
    public IQueryable<T> FindWhere(
        Expression<Func<T, bool>> predicate) {
        return _objectSet.Where(predicate);
    }
    public T FindById(int id) {
        return _objectSet.Single(o => o.Id == id);
    }
    public void Add(T newEntity) {
        _objectSet.AddObject(newEntity);
    }
    public void Remove(T entity) {
        _objectSet.DeleteObject(entity);
    }
    protected ObjectSet<T> _objectSet;
}

```

`IRepository<T>` 方法为我们提供对查询的更多控制，因为客户端必须调用方法才能访问实体。在方法中，我们可以提供额外的检查和 LINQ 运算符来强制应用程序约束。请注意，接口对泛型类型参数具有两个约束。第一个约束是 `ObjectSet<T>` 所需的类缺点破坏，第二个约束强制我们的实体实现 `IEntity` - 为应用程序创建的抽象。`IEntity` 接口强制实体具有可读 `Id` 属性，然后可以在 `FindById` 方法中使用此属性。`IEntity` 是用下面的代码定义的。

```

public interface IEntity {
    int Id { get; }
}

```

`IEntity` 可能被视为一小部分持久性无感知，因为需要实体才能实现此接口。请记住持久性无感知是关于权衡的，许多 `FindById` 功能将超过接口施加的约束。接口对可测试性没有影响。

实例化 live `IRepository<T>` 需要 EF4 `ObjectContext`，因此具体的工作单元实现应管理实例化。

```

public class SqlUnitOfWork : IUnitOfWork {
    public SqlUnitOfWork() {
        var connectionString =
            ConfigurationManager
                .ConnectionStrings[ConnectionStringName]
                .ConnectionString;

        _context = new ObjectContext(connectionString);
        _context.ContextOptions.LazyLoadingEnabled = true;
    }

    public IRepository<Employee> Employees {
        get {
            if (_employees == null) {
                _employees = new SqlRepository<Employee>(_context);
            }
            return _employees;
        }
    }

    public IRepository<TimeCard> TimeCards {
        get {
            if (_timeCards == null) {
                _timeCards = new SqlRepository<TimeCard>(_context);
            }
            return _timeCards;
        }
    }

    public void Commit() {
        _context.SaveChanges();
    }
}

SqlRepository<Employee> _employees = null;
SqlRepository<TimeCard> _timeCards = null;
readonly ObjectContext _context;
const string ConnectionStringName = "EmployeeDataModelContainer";
}

```

使用自定义存储库

使用自定义存储库并不大不同于使用基于 `IObjectSet<T>` 的存储库。我们首先需要调用一个存储库方法来获取 `IQueryable<T>` 引用，而不是直接对属性应用 LINQ 运算符。

```

public ViewResult Index() {
    var model = _repository.FindAll()
        .Include("TimeCards")
        .OrderBy(e => e.HireDate);
    return View(model);
}

```

请注意，我们先前实现的自定义 `Include` 运算符无需更改即可运行。存储库的 `FindById` 方法从尝试检索单个实体的操作中删除重复逻辑。

```

public ViewResult Details(int id) {
    var model = _repository.FindById(id);
    return View(model);
}

```

我们所检查的两种方法的可测试性并没有显著的差异。我们可以通过构建由 `HashSet<>` 员工支持的具体类来提供 `IRepository<T>` 的伪实现，就像我们在上一节中所做的那样。但是，某些开发人员更喜欢使用 mock 对象和模拟对象框架，而不是生成 fakes。我们将在下一节中介绍如何使用模拟来测试实现并讨论模拟和 fakes 之间的差异。

用模拟进行测试

可以使用不同的方法来构建 Fowler 调用 "测试双精度" 的情况。测试双精度型(如电影 stunt 双精度型)是在测试期间生成的、用于实际生产对象的对象。我们创建的内存中存储库是与 SQL Server 通信的存储库的测试。我们已了解如何在单元测试过程中使用这些测试双线来隔离代码并使测试运行速度更快。

我们生成的测试翻倍，实现了真实的工作实现。在幕后，每个对象都存储一个具体的对象集合，并将在测试过程中操作存储库时，在此集合中添加和移除对象。一些开发人员喜欢以这种方式生成测试，这种方法包括真实的代码和工作实现。这些测试是 *fakes* 的。它们具有有效的实现，但并不真正足以用于生产。伪存储库实际上不会写入数据库。伪 SMTP 服务器实际上不会通过网络发送电子邮件。

模拟与 Fakes

还有另一种类型的测试 double 称为 **模拟**。尽管 fakes 具有有效的实现，但模拟没有实现。使用 mock 对象框架的帮助，我们在运行时构造这些模拟对象并将其用作测试双精度型。在本部分中，我们将使用开源模拟 framework Moq。下面是一个简单的示例，说明如何使用 Moq 为员工存储库动态创建测试 double。

```
Mock< IRepository< Employee >> mock =
    new Mock< IRepository< Employee >>();
IRepository< Employee > repository = mock.Object;
repository.Add(new Employee());
var employee = repository.FindById(1);
```

我们要求 Moq **IRepository< 员工 >** 实现，并且它会动态生成一个。通过访问 **Mock< T >** 对象的属性，我们可以访问实现 **IRepository** 的对象<员工>。我们可以将此内部对象传递到我们的控制器中，并不知道这是测试 double 还是真正的存储库。可以调用对象上的方法，就像我们使用实际实现调用对象上的方法一样。

当我们调用 Add 方法时，你必须知道 mock 存储库将执行的操作。由于 mock 对象尚无实现，因此不执行任何操作。幕后没有具体的集合，就像我们编写的 fakes 一样，因此放弃了该员工。FindById 的返回值怎么样？在这种情况下，mock 对象执行的唯一操作就是返回默认值。由于我们返回引用类型(雇员)，因此返回值为 null 值。

模拟可能会毫无意义。但是，我们还没有谈到模拟的两个功能。首先，Moq 框架记录模拟对象上发出的所有调用。稍后在代码中，我们可以询问 Moq 是否有人调用了 Add 方法，或者有人调用了 FindById 方法。稍后我们将介绍如何在测试中使用此 "黑色框" 录制功能。

第二大功能是，我们可以使用 Moq 来为模拟对象编程。预期会告诉 mock 对象如何响应任何给定的交互。例如，我们可以将预期计划为模拟，并告诉它在用户调用 FindById 时返回 employee 对象。Moq 框架使用安装 API 和 lambda 表达式来对这些预期进行编程。

```
[TestMethod]
public void MockSample() {
    Mock< IRepository< Employee >> mock =
        new Mock< IRepository< Employee >>();
    mock.Setup(m => m.FindById(5))
        .Returns(new Employee { Id = 5 });
    IRepository< Employee > repository = mock.Object;
    var employee = repository.FindById(5);
    Assert.IsTrue(employee.Id == 5);
}
```

在此示例中，我们要求 Moq 动态构建一个存储库，然后对存储库进行计划。当有人调用 FindById 方法传递值5时，预期会告知 mock 对象返回 Id 值为5的新 employee 对象。此测试通过，我们不需要生成完整的实现来实现< T >的虚假 IRepository。

让我们重新访问之前编写的测试，并对其进行返工，使用模拟而不是 fakes。就像以前一样，我们将使用基类来设置控制器所有测试所需的基础结构的公共部分。

```

public class EmployeeControllerTestBase {
    public EmployeeControllerTestBase() {
        _employeeData = EmployeeObjectMother.CreateEmployees()
            .AsQueryable();
        _repository = new Mock< IRepository<Employee>>();
        _unitOfWork = new Mock< IUnitOfWork>();
        _unitOfWork.Setup(u => u.Employees)
            .Returns(_repository.Object);
        _controller = new EmployeeController(_unitOfWork.Object);
    }

    protected IQueryable<Employee> _employeeData;
    protected Mock< IUnitOfWork> _unitOfWork;
    protected EmployeeController _controller;
    protected Mock< IRepository<Employee>> _repository;
}

```

安装程序代码的基本保持不变。我们将使用 Moq 来构造 mock 对象，而不是使用 fakes。当代码调用 Employees 属性时，该基类将排列模拟工作单元以返回 mock 存储库。模拟设置的其余部分将在专用于每个特定方案的测试装置内进行。例如，当操作调用 mock 存储库的 FindAll 方法时，用于索引操作的测试装置将设置 mock 存储库以返回员工列表。

```

[TestClass]
public class EmployeeControllerIndexActionTests
    : EmployeeControllerTestBase {
    public EmployeeControllerIndexActionTests() {
        _repository.Setup(r => r.FindAll())
            .Returns(_employeeData);
    }
    // .. tests
    [TestMethod]
    public void ShouldBuildModelWithAllEmployees() {
        var result = _controller.Index();
        var model = result.ViewData.Model
            as IEnumerable<Employee>;
        Assert.IsTrue(model.Count() == _employeeData.Count());
    }
    // .. and more tests
}

```

除了预期之外，我们的测试看起来类似于以前的测试。然而，使用模拟框架的录制功能，我们可以从不同角度进行测试。在下一部分中，我们将介绍这个新的透视。

状态与交互测试

你可以使用不同的技术来测试具有 mock 对象的软件。一种方法是使用基于状态的测试，这是我们目前在本文中所执行的操作。基于状态的测试对软件的状态进行断言。在最后一次测试中，我们在控制器上调用了操作方法，并对它应生成的模型作出了一个断言。下面是测试状态的一些其他示例：

- 创建 "创建" 后，验证存储库是否包含新的 employee 对象。
- 验证该模型在执行 Index 后是否包含所有员工的列表。
- 验证 "删除" 执行后存储库不包含给定的员工。

模拟对象的另一种方法是验证交互。虽然基于状态的测试对对象的状态进行断言，但基于交互的测试会使断言与对象交互的方式有关。例如：

- 验证在创建执行时控制器是否调用存储库的 Add 方法。
- 当执行索引时，验证控制器是否调用存储库的 FindAll 方法。
- 验证控制器是否调用工作单元的 Commit 方法，以便在执行编辑时保存更改。

交互测试通常需要较少的测试数据，因为我们不会在集合中闲逛和验证计数。例如，如果我们知道详细信息操作将使用正确的值调用存储库的 `FindById` 方法，则操作可能会正常运行。我们可以验证此行为，而无需设置任何要从 `FindById` 返回的测试数据。

```
[TestClass]
public class EmployeeControllerDetailsActionTests
    : EmployeeControllerTestBase {
    // ...
    [TestMethod]
    public void ShouldInvokeRepositoryToFindEmployee() {
        var result = _controller.Details(_detailsId);
        _repository.Verify(r => r.FindById(_detailsId));
    }
    int _detailsId = 1;
}
```

上述测试装置中唯一需要的设置是由基类提供的设置。当我们调用控制器操作时，Moq 将记录与 mock 存储库的交互。使用 Moq 的 Verify API，如果控制器使用正确的 ID 值调用 `FindById`，则可以询问 Moq。如果控制器未调用方法，或调用方法时出现意外的参数值，则 Verify 方法将引发异常，并且测试将失败。

下面是另一个示例，用于验证创建操作对当前工作单元调用 `Commit`。

```
[TestMethod]
public void ShouldCommitUnitOfWork() {
    _controller.Create(_newEmployee);
    _unitOfWork.Verify(u => u.Commit());
}
```

交互测试有一个危险，就是指定交互。Mock 对象记录和验证每个与 mock 对象的交互的能力并不意味着测试应尝试验证每个交互。某些交互是实现详细信息，只应验证满足当前测试所需的交互。

模拟或 fakes 之间的选择主要取决于你要测试的系统和个人(或团队)首选项。Mock 对象可以极大地减少实现测试双精度值所需的代码量，但并非每个人都能得心应手地进行编程和验证交互。

结论

在本文中，我们演示了几种创建可测试代码的方法，同时使用 ADO.NET 实体框架来实现数据持久性。我们可以利用内置抽象，例如 `IObjectSet<T>`，或创建自己的抽象，例如 `IRepository<T>`。在这两种情况下，ADO.NET 中的 POCO 支持实体框架 4.0，使这些抽象的使用者能够维持永久性未知和高度可测试性。隐式延迟加载等附加 EF4 功能允许业务和应用程序服务代码工作，而无需担心关系数据存储的详细信息。最后，我们创建的抽象可以轻松地在单元测试中模拟或伪造，而且我们可以使用这些测试来实现快速运行、高度隔离和可靠的测试。

其他资源

- Robert, "[单责任原则](#)"
- 圣马丁 Fowler, [从企业应用程序体系结构模式的模式目录](#)
- Griffin Caprio, "[依赖关系注入](#)"
- 数据可编程性博客, "[演练: 测试驱动开发与实体框架 4.0](#)"。
- 数据可编程性博客, "[使用存储库和实体框架4.0 的工作单元模式](#)"
- Aaron Jensen, "[机器规格简介](#)"
- Eric 先生, "[BDD With MSTest](#)"
- Eric Evans, "[域驱动设计](#)"
- 圣马丁 Fowler, "[模拟不是存根](#)"
- 圣马丁 Fowler, "[Test Double](#)"
- [Moq](#)

事迹

Scott Allen 是技术人员在 Pluralsight 和创始人 OdeToCode.com 的成员。在15年的商业软件开发中, Scott 已经处理了从8位嵌入式设备到高度可扩展的 ASP.NET web 应用程序的各种解决方案。你可以在 OdeToCode 上或在 Twitter 的<https://twitter.com/OdeToCode>上联系 Scott。

创建模型

2020/4/8 •

EF 模型可存储有关应用程序类和属性如何映射到数据库表和列的详细信息。创建 EF 模型有两种主要方式：

- **使用 Code First**: 开发者编写代码来指定模型。EF 基于实体类和开发者提供的其他模型配置，在运行时生成模型和映射。
- **使用 EF 设计器**: 开发者使用 EF 设计器进行绘制以指定模型。生成的模型以 XML 格式存储在具有 EDMX 扩展名的文件中。应用程序域对象通常从概念模型中自动生成。

EF 工作流

这两种方式都可用于定位现有数据库或创建新数据库，最终会产生 4 种不同的工作流。了解哪一种最适合：

—	使用 Code First 在代码中定义模型，然后生成数据库。	通过 Model First 使用框和线定义模型，然后生成数据库。
—	使用 Code First 创建映射到现有数据库的基于代码的模型。	使用 Database First 创建映射到现有数据库的框和线模型。

观看视频：我应该使用哪种 EF 工作流？

这段短片介绍了其中的差异，以及如何找到适合的工作流。

主讲人 : Rowan Miller



[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

如果在观看视频后仍无法顺利决定使用 EF 设计器还是 Code First，那么分别请了解两者！

查看二者的本质区别

无论是使用 Code First 还是 EF 设计器，EF 模型始终具有以下几个组件：

- 应用程序域对象或实体类型本身。这通常称为对象层
- 使用**实体数据模型**描述的，由特定于域的实体类型和关系组成的概念模型。该层通常用字母“C”表示概念。
- 表示数据库中定义的表、列和关系的存储模型。该层通常用字母“S”表示存储。
- 概念模型和数据库架构之间的映射。该映射通常称为“C-S”映射。

EF 的映射引擎利用“C-S”映射将针对实体的操作（例如创建、读取、更新和删除）转换为针对数据库中的表的等效操作。

概念模型和应用程序的对象之间的映射通常称为“O-C”映射。与“C-S”映射相比，“O-C”映射是一对一的隐式映射：概念模型中定义的实体、属性和关系需要与 .NET 对象的形状和类型相匹配。从 EF4 及更高版本开始，对象层可以由具有属性的简单对象组成，无需任何 EF 依赖关系。这些通常称为简单传统 CLR 对象 (POCO)，同时类型和属性映射以名称匹配约定为基础进行。以前，在 EF 3.5 中，对象层存在特定限制，例如实体必须派生自 EntityObject

类，并且必须带有 EF 属性以实现“O-C”映射。

Code First 到新数据库

2020/3/12 •

此视频和分步演练提供了面向新数据库的 Code First 开发的简介。此方案包括针对不存在的数据库和 Code First 将创建的数据库，或者 Code First 将向其中添加新表的空数据库。Code First 允许使用 C# 或 VB.NET 类定义模型。还可以选择使用类和属性上的属性或使用 Fluent API 来执行其他配置。

观看视频

此视频介绍了面向新数据库的 Code First 开发。此方案包括针对不存在的数据库和 Code First 将创建的数据库，或者 Code First 将向其中添加新表的空数据库。Code First 允许使用 C# 或 VB.NET 类定义模型。还可以选择使用类和属性上的属性或使用 Fluent API 来执行其他配置。

主讲人 : [Rowan Miller](#)

视频 : [WMV MP4](#) | [WMV \(ZIP\)](#)

先决条件

需要至少安装 Visual Studio 2010 或 Visual Studio 2012 才能完成此演练。

如果你使用的是 Visual Studio 2010，你还需要安装 [NuGet](#)。

1. 创建应用程序

为了简单起见，我们将构建一个使用 Code First 执行数据访问的基本控制台应用程序。

- 打开 Visual Studio
- 文件->> 项目。
- 从左侧菜单和控制台应用程序选择 Windows
- 输入 **CodeFirstNewDatabaseSample** 作为名称
- 选择“确定”

2. 创建模型

让我们使用类定义一个非常简单的模型。我们只是在 Program.cs 文件中对其进行定义，但在实际应用程序中，您可以将类拆分为单独的文件，并可能是单独的项目。

在 Program.cs 中的 Program 类定义下，添加以下两个类。

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}

```

你会注意到，我们正在建立两个导航属性（博客和博客）虚拟。这将启用实体框架的延迟加载功能。延迟加载是指在您尝试访问这些属性时，这些属性的内容将自动从数据库加载。

3. 创建上下文

现在可以定义一个派生上下文，它表示与数据库的会话，从而使我们能够查询和保存数据。我们定义了从 DbContext 派生的上下文，并为模型中的每个类公开了类型化的 DbSet< TEntity >。

现在，我们开始使用实体框架中的类型，因此我们需要添加 EntityFramework NuGet 包。

- 项目-> 管理 NuGet 程序包 。注意：如果你没有 "管理 NuGet 包 ..." 选项你应安装[最新版本的 NuGet](#)
- 选择 "联机" 选项卡
- 选择 EntityFramework 包
- 单击"安装"

在 Program.cs 的顶部为 system.string 添加 using 语句。

```
using System.Data.Entity;
```

在 Program.cs 中的 Post 类下面添加以下派生上下文。

```

public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}

```

下面是 Program.cs 应该包含的完整列表。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data.Entity;

namespace CodeFirstNewDatabaseSample
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }

        public virtual List<Post> Posts { get; set; }
    }

    public class Post
    {
        public int PostId { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        public int BlogId { get; set; }
        public virtual Blog Blog { get; set; }
    }

    public class BloggingContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
        public DbSet<Post> Posts { get; set; }
    }
}
```

这就是我们开始存储和检索数据所需的全部代码。很显然，幕后会出现很多情况，接下来我们将对此进行介绍，但首先我们来看看这一点。

4. 读取 & 写入数据

在 Program.cs 中实现 Main 方法，如下所示。此代码创建一个新的上下文实例，然后使用它来插入新的博客。然后，它使用 LINQ 查询从按标题字母顺序排序的数据库中检索所有博客。

```

class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

你现在可以运行该应用程序并对其进行测试。

```

Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...

```

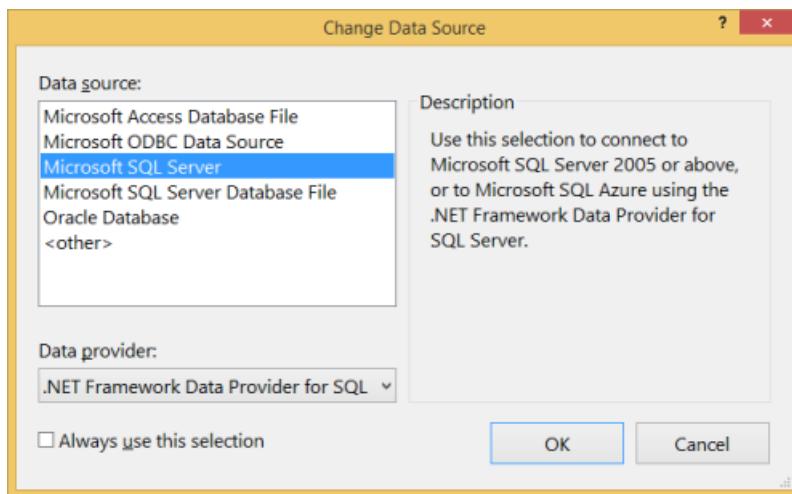
我的数据在哪里？

按惯例，`DbContext` 已为您创建了一个数据库。

- 如果本地 SQL Express 实例可用（默认情况下随 Visual Studio 2010 一起安装），则 Code First 已在该实例上创建了数据库
- 如果 SQL Express 不可用，Code First 将尝试使用 LocalDB（默认情况下安装 Visual Studio 2012）
- 数据库以派生上下文的完全限定名称命名，在本例中为 `CodeFirstNewDatabaseSample`。
"bloggingcontext"

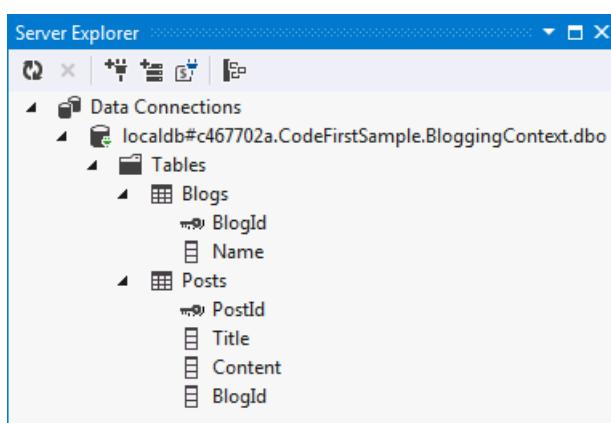
这些只是默认的约定，有多种方法可用于更改 Code First 使用的数据库，有关详细信息，请访问 [DbContext 如何发现模型和数据库连接主题](#)。你可以使用 Visual Studio 中的服务器资源管理器连接到此数据库

- 视图-> 服务器资源管理器
- 右键单击 "数据连接"，然后选择 "添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择 Microsoft SQL Server 作为数据源



- 连接到 LocalDB 或 SQL Express, 具体取决于你安装的是哪个

现在, 我们可以检查 Code First 创建的架构。



DbContext 通过查看我们定义的 DbSet 属性来确定要包括在模型中的类。然后, 它使用默认的 Code First 约定集来确定表和列的名称、确定数据类型、查找主键等。稍后在本演练中, 我们将介绍如何覆盖这些约定。

5. 处理模型更改

现在, 可以对模型进行一些更改, 当我们进行这些更改时, 我们还需要更新数据库架构。为此, 我们将使用一项称为 Code First 迁移的功能, 或使用较短的迁移。

迁移允许我们提供一组有序的步骤, 这些步骤描述如何升级(和降级)我们的数据库架构。其中的每个步骤(称为迁移)都包含一些描述要应用的更改的代码。

第一步是为 "Bloggingcontext" 启用 Code First 迁移。

- 工具-> 库包管理器-> 程序包管理器控制台
- 在包管理器控制台中运行 Enable-Migrations 命令
- 已将新的迁移文件夹添加到包含两个项目的项目中:
 - Configuration.cs –此文件包含迁移将用于迁移 "bloggingcontext" 的设置。对于本演练, 我们不需要更改任何内容, 但可以在此处指定种子数据, 注册其他数据库的提供程序, 更改将在其中生成迁移的命名空间等。
 - <时间戳>_InitialCreate.cs –这是第一次迁移, 它表示已应用到数据库的更改, 以使其成为包含博客和文章表的数据库的空数据库。尽管我们允许 Code First 自动为我们创建这些表, 但现在我们已选择迁移, 它们已转换为迁移。还在本地数据库中记录了已应用此迁移的 Code First。文件名上的时间戳用于排序目的。

现在, 我们来对模型进行更改, 将 Url 属性添加到博客类:

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public virtual List<Post> Posts { get; set; }
}

```

- 在 Package Manager Console 中运行**Add-迁移 AddUrl**命令。添加迁移命令将检查自上次迁移后发生的更改，并基架找到的任何更改进行新迁移。我们可以为迁移指定一个名称;在此示例中，我们将调用迁移"AddUrl"。基架代码是说，我们需要向 dbo 添加 Url 列，该 Url 列可以保存字符串数据。博客表。如果需要，我们可以编辑基架代码，但在这种情况下不需要这样做。

```

namespace CodeFirstNewDatabaseSample.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

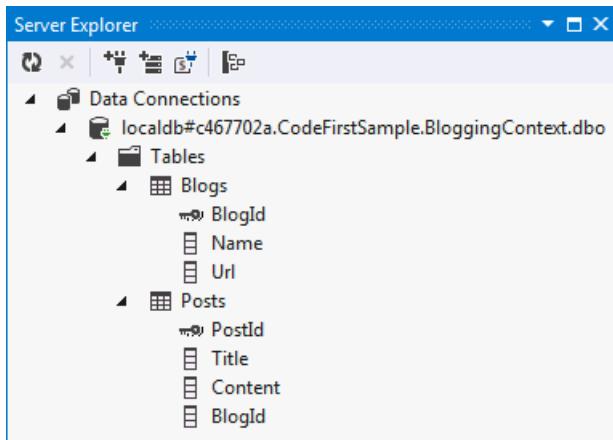
    public partial class AddUrl : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Blogs", "Url", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Blogs", "Url");
        }
    }
}

```

- 在 Package Manager Console 中运行 "**更新数据库**" 命令。此命令会将任何挂起的迁移应用到数据库。已应用 InitialCreate 迁移，因此迁移只会应用新的 AddUrl 迁移。提示：调用 "更新数据库" 时，可以使用 -Verbose 开关来查看正在对数据库执行的 SQL。

此时，新的 Url 列将添加到数据库的 "博客" 表中：



6. 数据注释

到目前为止，我们只是让 EF 使用其默认约定来发现模型，但有时我们的类不遵循约定，我们需要能够进一步进行配置。有两个选项可供选择：我们将在本节中查看数据批注，然后在下一部分中查看 Fluent API。

- 让我们向模型中添加一个用户类

```
public class User
{
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- 我们还需要向派生的上下文添加一个集

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }
}
```

- 如果我们尝试添加迁移，会收到一个错误，指出 "EntityType" User "未定义键。为此 EntityType 定义密钥。" 因为 EF 无法知道用户名应为用户的主密钥。
- 现在我们将使用数据批注，因此我们需要在 Program.cs 的顶部添加 using 语句

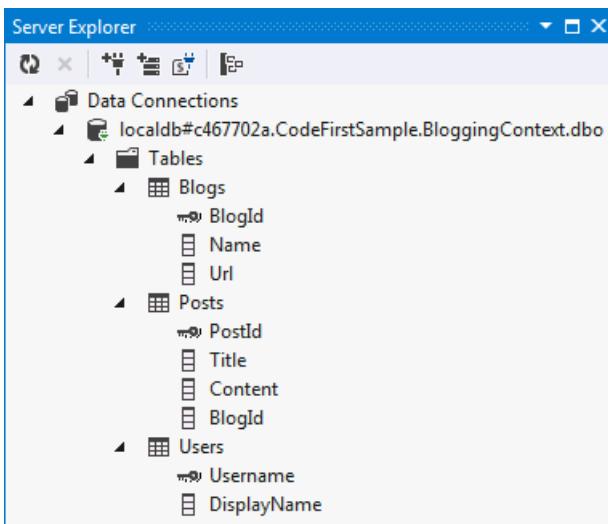
```
using System.ComponentModel.DataAnnotations;
```

- 现在为用户名属性添加批注，以确定它是主键

```
public class User
{
    [Key]
    public string Username { get; set; }
    public string DisplayName { get; set; }
}
```

- 使用Add-迁移 AddUser命令基架迁移，将这些更改应用到数据库
- 运行 "更新-数据库" 命令以将新迁移应用到数据库

新表现在已添加到数据库中：



EF 支持的全部批注列表为：

- [KeyAttribute](#)
- [StringLengthAttribute](#)
- [MaxLengthAttribute](#)

- [ConcurrencyCheckAttribute](#)
- [有 requiredattribute](#)
- [TimestampAttribute](#)
- [ComplexTypeAttribute](#)
- [ColumnAttribute](#)
- [TableAttribute](#)
- [InversePropertyAttribute](#)
- [ForeignKeyAttribute](#)
- [DatabaseGeneratedAttribute](#)
- [NotMappedAttribute](#)

7. Fluent API

在上一部分中，我们介绍了如何使用数据批注来补充或替代约定检测到的内容。配置模型的另一种方法是通过 Code First Fluent API。

大多数模型配置可以使用简单地数据批注完成。Fluent API 是一种更高级的方法，它指定模型配置，该配置涵盖了数据批注可以执行的所有操作以及数据批注无法实现的更高级配置。数据批注和 Fluent API 可一起使用。

若要访问 Fluent API 可以重写 DbContext 中的 OnModelCreating 方法。假设我们要将用户名栏存储在中的列重命名为显示_名称。

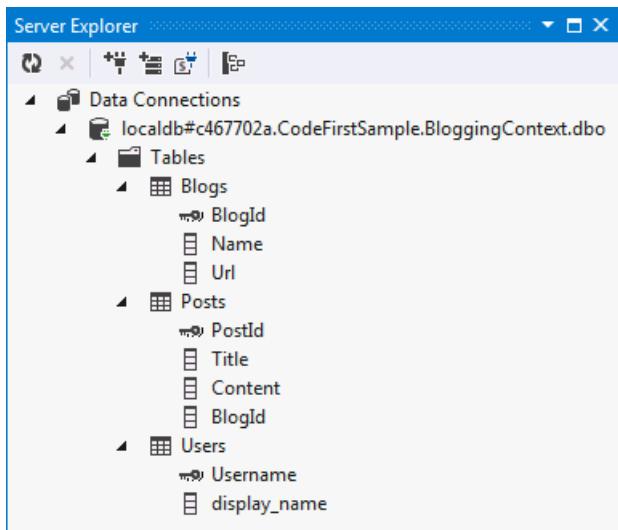
- 用以下代码替代 "Bloggingcontext" 上的 OnModelCreating 方法

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
            .Property(u => u.DisplayName)
            .HasColumnName("display_name");
    }
}
```

- 使用Add-迁移 ChangeDisplayName命令基架迁移，将这些更改应用到数据库。
- 运行 "更新-数据库" 命令，以将新迁移应用到数据库。

DisplayName 列现在已重命名为显示_名称：



Summary

在本演练中，我们将使用新数据库查看 Code First 开发。我们使用类定义了一个模型，然后使用该模型来创建数据库并存储和检索数据。创建数据库后，我们使用 Code First 迁移来更改模型，因为模型已演化。我们还了解了如何使用数据批注和熟知的 API 配置模型。

Code First 到现有数据库

2020/3/11 •

此视频和分步演练提供了面向现有数据库 Code First 开发的简介。Code First 允许使用 C# 或 VB.NET 类定义模型。另外，还可以使用类和属性上的属性或使用 Fluent API 来执行其他配置。

观看视频

此视频[现在适用于第9频道](#)。

先决条件

你将需要安装Visual Studio 2012或Visual Studio 2013才能完成此演练。

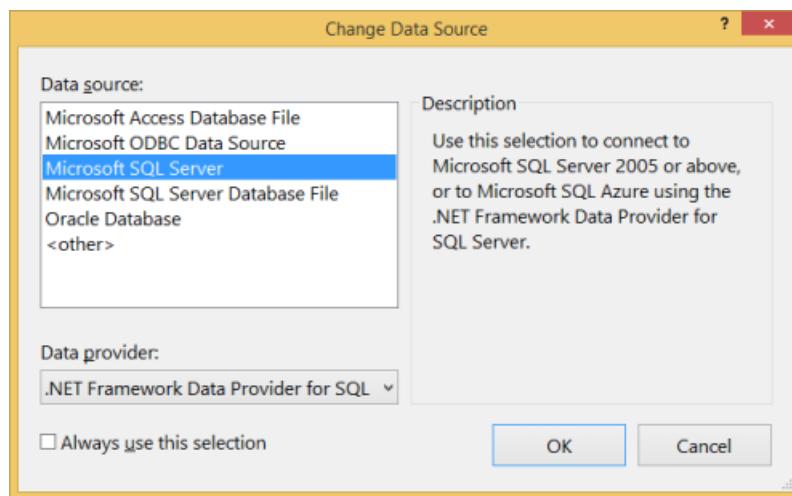
还需要安装适用于 Visual Studio 的 Entity Framework Tools 6.1（或更高版本）。有关安装最新版本的 Entity Framework Tools 的信息，请参阅[获取实体框架](#)。

1. 创建现有数据库

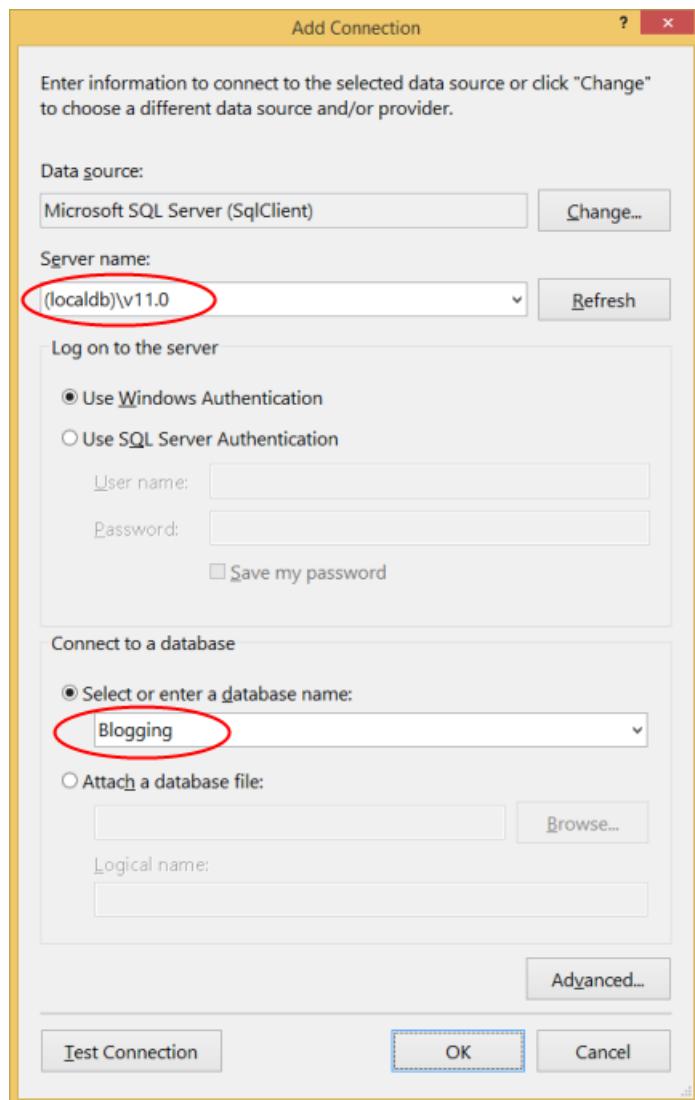
通常，当目标为现有数据库时，它将被创建，但在本演练中，我们需要创建一个要访问的数据库。

接下来，生成数据库。

- 打开 Visual Studio
- 视图-> 服务器资源管理器
- 右键单击 "数据连接-> 添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择Microsoft SQL Server作为数据源



- 连接到 LocalDB 实例，并输入博客作为数据库名称



- 选择“确定”，系统会询问您是否要创建新数据库，请选择“是”



- 新数据库现在将出现在服务器资源管理器中，右键单击该数据库并选择“新建查询”
- 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择“执行”

```

CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId]) ON
DELETE CASCADE
);

INSERT INTO [dbo].[Blogs] ([Name],[Url])
VALUES ('The Visual Studio Blog', 'http://blogs.msdn.com/visualstudio/')

INSERT INTO [dbo].[Blogs] ([Name],[Url])
VALUES ('.NET Framework Blog', 'http://blogs.msdn.com/dotnet/')

```

2. 创建应用程序

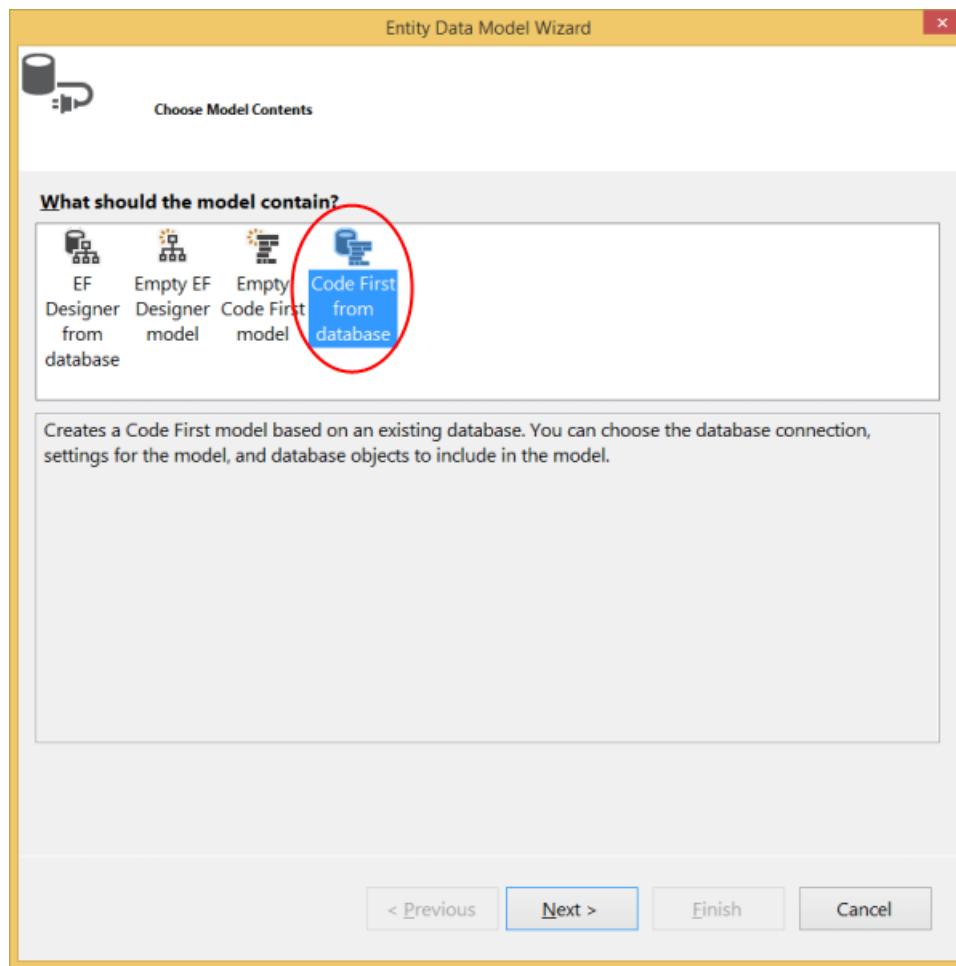
为简单起见，我们将构建一个使用 Code First 进行数据访问的基本控制台应用程序：

- 打开 Visual Studio
- 文件->> 项目。
- 从左侧菜单和控制台应用程序选择Windows
- 输入CodeFirstExistingDatabaseSample作为名称
- 选择“确定”

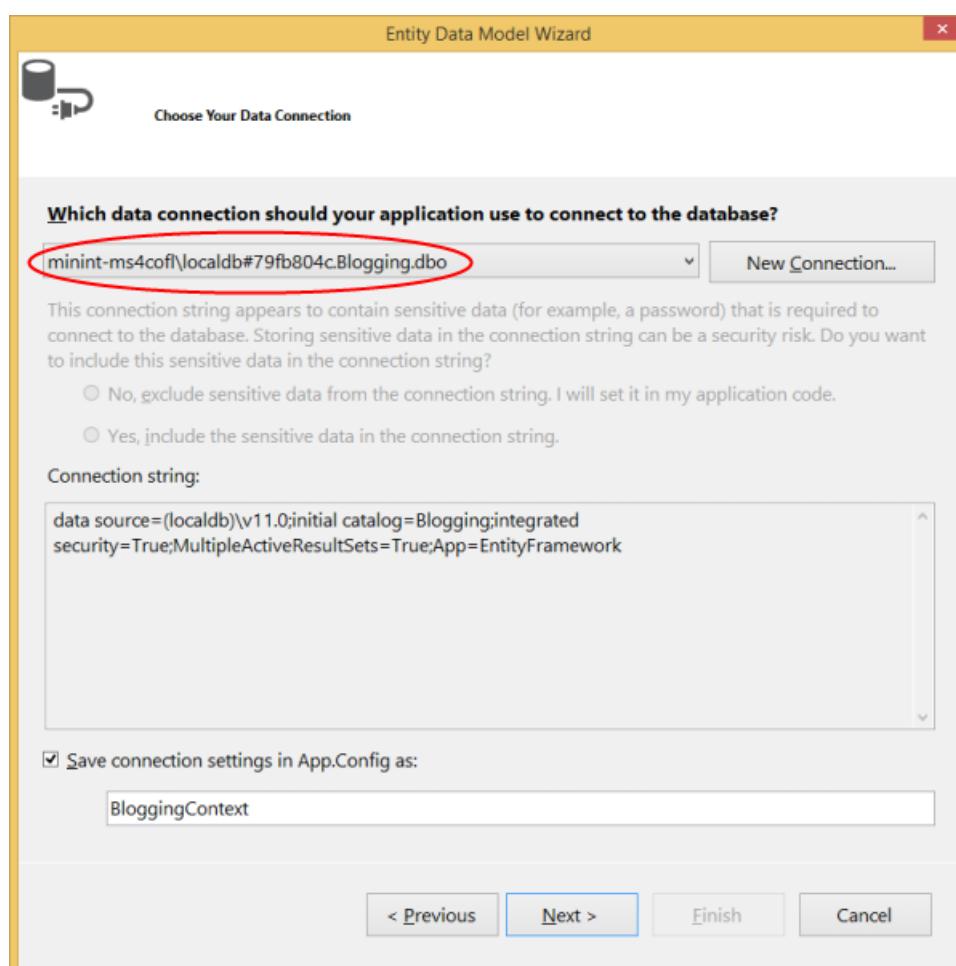
3. 反向工程模型

我们将使用适用于 Visual Studio 的 Entity Framework Tools 帮助我们生成一些用于映射到数据库的初始代码。这些工具只是生成代码，你也可以根据需要手动键入代码。

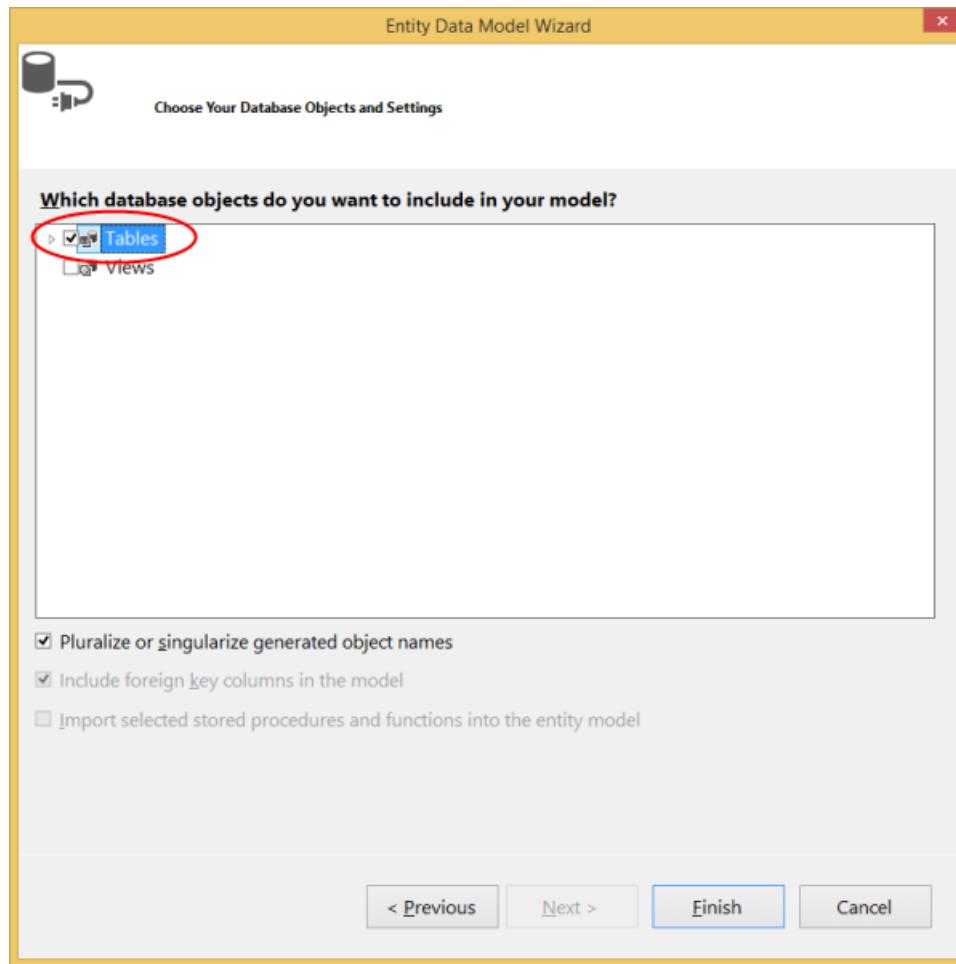
- 项目-> "添加新项 ..."
- 从左侧菜单中选择 "数据"，然后ADO.NET 实体数据模型
- 输入 "bloggingcontext" 作为名称，然后单击 "确定"
- 这将启动实体数据模型向导
- 选择 "从数据库 Code First "，然后单击 "下一步"



- 选择与在第一部分中创建的数据库的连接, 然后单击 "下一步"



- 单击 "表" 旁边的复选框以导入所有表, 然后单击 "完成"



反向工程过程完成后，会向项目中添加大量项目，让我们看看添加的内容。

配置文件

已将 App.config 文件添加到项目中，此文件包含现有数据库的连接字符串。

```
<connectionStrings>
  <add
    name="BlogginContext"
    connectionString="data source=(localdb)\mssqllocaldb;initial catalog=Bloggin;integrated
    security=True;MultipleActiveResultSets=True;App=EntityFramework"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

你还会注意到配置文件中的其他一些设置，这些是默认的 EF 设置，告诉 Code First 在何处创建数据库。由于我们要映射到现有数据库，因此在应用程序中将忽略这些设置。

派生上下文

已将 "bloggingcontext" 类添加到项目。上下文表示与数据库的会话，从而使我们能够查询并保存数据。上下文公开模型中每个类型的 DbSet< TEntity >。你还会注意到，默认构造函数使用名称 = 语法调用基构造函数。这会告知 Code First 应从配置文件加载要用于此上下文的连接字符串。

```
public partial class BloggingContext : DbContext
{
    public BloggingContext()
        : base("name=BloggingContext")
    {
    }

    public virtual DbSet<Blog> Blogs { get; set; }
    public virtual DbSet<Post> Posts { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
    }
}
```

使用配置文件中的连接字符串时，应始终使用`name =*` 语法。这可以确保在连接字符串不存在时，实体框架将引发，而不是按约定创建新数据库。*

模型类

最后，还将**博客**和**Post**类添加到了项目中。这些是构成模型的域类。你将看到应用于类的数据批注，用于指定 Code First 约定不与现有数据库的结构一致的配置。例如，你将在**Blog.Name**和**Blog**上看到**StringLength**批注，因为它们在数据库中的最大长度为200（Code First 默认值是使用 SQL Server）中的数据库提供程序（**nvarchar(max)**）支持的多长度。

```
public partial class Blog
{
    public Blog()
    {
        Posts = new HashSet<Post>();
    }

    public int BlogId { get; set; }

    [StringLength(200)]
    public string Name { get; set; }

    [StringLength(200)]
    public string Url { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}
```

4. 读取 & 写入数据

现在，我们有了一个模型，可以使用它来访问某些数据了。在**Program.cs**中实现**Main**方法，如下所示。此代码创建一个新的上下文实例，然后使用它来插入新的**博客**。然后，它使用 LINQ 查询从按标题字母顺序排序的数据库中检索所有**博客**。

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.WriteLine("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

现在可以运行应用程序并对其进行测试。

```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
.NET Framework Blog
ADO.NET Blog
The Visual Studio Blog
Press any key to exit...
```

如果数据库发生了更改，该怎么办？

"Code First 到数据库" 向导旨在生成一组可以进行调整和修改的起始点。如果数据库架构发生更改，则可以手动编辑类，也可以执行其他反向工程来覆盖类。

使用现有数据库的 Code First 迁移

如果要对现有数据库使用 Code First 迁移，请参阅[Code First 迁移到现有数据库](#)。

摘要

在本演练中，我们使用现有数据库查看 Code First 开发。我们使用了 Visual Studio Entity Framework Tools 来反向工程映射到数据库并可用于存储和检索数据的一组类。

Code First 数据批注

2020/3/11 •

NOTE

■ ef 4.1 -在实体框架4.1 中引入了本页中所述的功能、api 等。如果你使用的是早期版本，则不会应用此信息中的部分或全部。

此页面上的内容适用于最初由 Julie Lerman (<<http://thedatafarm.com>>) 编写的文章。

利用实体框架 Code First，你可以使用自己的域类来表示 EF 依赖来执行查询、更改跟踪和更新功能的模型。Code First 利用称为“约定 over 配置”的编程模式。Code First 将假设你的类遵循实体框架的约定，在这种情况下，将自动处理如何执行其作业。但是，如果你的类不遵循这些约定，则可以将配置添加到你的类，以便为 EF 提供必需的信息。

Code First 提供了向你的类添加这些配置的两种方法。其中一种方法是使用名为 DataAnnotations 的简单特性，第二种方法是使用 Code First 的流畅 API，这为你提供了一种在代码中以强制方式描述配置的方式。

本文重点介绍如何使用 DataAnnotations (DataAnnotations 命名空间中的 System.ComponentModel) 来配置类-突出显示最常用的配置。DataAnnotations 也可由许多 .NET 应用程序(如 ASP.NET MVC)理解，这允许这些应用程序利用相同的注释进行客户端验证。

模型

我将使用简单的类对 Code First DataAnnotations 进行演示：博客和文章。

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

正如他们一样，博客和文章类可以方便地遵循 code first 约定，无需进行调整即可启用 EF 兼容性。但是，您还可以使用批注向 EF 提供有关它们所映射到的类和数据库的详细信息。

Key

实体框架依赖于每个实体，每个实体都有一个用于实体跟踪的键值。Code First 的一种约定是隐式键属性；Code First 将查找名为“Id”的属性，或者查找类名称和“Id”（如“BlogId”）的组合。此属性将映射到数据库中的主键列。

博客和帖子类都遵循此约定。如果不是，怎么办？如果博客使用的是名称 `PrimaryTrackingKey`（甚至是 `foo`），该怎么办？如果代码优先找不到与此约定相匹配的属性，则会引发异常，因为实体框架要求必须有一个键属性。你可以使用密钥批注来指定要用作 `EntityKey` 的属性。

```
public class Blog
{
    [Key]
    public int PrimaryTrackingKey { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}
```

如果使用代码优先的数据生成功能，博客表将具有名为 `PrimaryTrackingKey` 的主键列，默认情况下，此列也定义为标识。

```
dbo.Blogs
Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), null)
BloggerName (nvarchar(128), null)
```

组合键

实体框架支持组合键-由多个属性组成的主键。例如，你可以有一个 `Passport` 类，其 primary key 是 `PassportNumber` 和 `IssuingCountry` 的组合。

```
public class Passport
{
    [Key]
    public int PassportNumber { get; set; }
    [Key]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

尝试在 EF 模型中使用以上类将导致 `InvalidOperationException`：

无法确定类型 "Passport" 的组合键。使用 `ColumnAttribute` 或 `HasKey` 方法为组合主键指定顺序。

若要使用组合键，实体框架要求您定义键属性的顺序。为此，可以使用列批注指定顺序。

NOTE

顺序值是相对的（而不是基于索引的），因此可以使用任何值。例如，100和200可接受而不是1和2。

```
public class Passport
{
    [Key]
    [Column(Order=1)]
    public int PassportNumber { get; set; }
    [Key]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }
    public DateTime Issued { get; set; }
    public DateTime Expires { get; set; }
}
```

如果有包含复合外键的实体，则必须指定用于相应主键属性的相同列排序。

只有外键属性中的相对顺序需要相同，分配给Order的确切值不需要匹配。例如，在下面的类中，可以使用3和4代替1和2。

```
public class PassportStamp
{
    [Key]
    public int StampId { get; set; }
    public DateTime Stamped { get; set; }
    public string StampingCountry { get; set; }

    [ForeignKey("Passport")]
    [Column(Order = 1)]
    public int PassportNumber { get; set; }

    [ForeignKey("Passport")]
    [Column(Order = 2)]
    public string IssuingCountry { get; set; }

    public Passport Passport { get; set; }
}
```

必需

必需的批注告诉 EF 需要特定属性。

向 Title 属性添加所需的将强制 EF (和 MVC) 确保属性中包含数据。

```
[Required]
public string Title { get; set; }
```

如果应用程序中没有额外的代码或标记更改，MVC 应用程序将执行客户端验证，甚至使用属性和批注名称动态生成消息。

Create

Blog

Title
 The Title field is required.

BloggerName

必需的属性还会影响生成的数据库，方法是将映射的属性设为不可为 null。请注意，Title 字段已更改为 "not null"。

NOTE

在某些情况下，即使属性是必需的，也无法使数据库中的列不可为 null。例如，对多个类型使用 TPH 继承战略数据时，会将其存储在一个表中。如果派生的类型包含所需的属性，则列不能为 null，因为并不是层次结构中的所有类型都具有此属性。

		dbo.Blogs
		Columns
		PrimaryTrackingKey (PK, int, not null)
		Title (nvarchar(128), not null)
		BloggerName (nvarchar(128), null)

MaxLength 和 MinLength

MaxLength 和 MinLength 属性允许您指定附加的属性验证，就像您在需要时所做的那样。

下面是具有长度要求的 BloggerName。该示例还演示了如何合并特性。

```
[MaxLength(10),MinLength(5)]  
public string BloggerName { get; set; }
```

MaxLength 批注将属性的长度设置为10，将影响数据库。

		Columns
		PrimaryTrackingKey (PK, int, not null)
		Title (nvarchar(128), not null)
		BloggerName (nvarchar(10), null)

MVC 客户端批注和 EF 4.1 服务器端批注都将接受此验证，并再次动态生成错误消息：“字段 BloggerName 必须是最大长度为“10”的字符串或数组类型。”该消息只需很长时间。许多批注允许您使用 ErrorMessage 特性指定错误消息。

```
[MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]  
public string BloggerName { get; set; }
```

您还可以在所需的批注中指定 ErrorMessage。

Create

Blog

Title	<input type="text" value="A New Blog"/>
BloggerName	<input type="text" value="Julie the Blogger"/> BloggerName must be 10 characters or less
<input type="button" value="Create"/>	

NotMapped

Code first 约定规定每个属于受支持数据类型的属性都在数据库中表示。但在应用程序中并不总是如此。例如，你可能在博客类中有一个属性，该属性基于“标题”和“BloggerName”字段创建代码。该属性可以动态创建，不需要存储。可以用 NotMapped 批注（如此 BlogCode 属性）来标记不映射到数据库的任何属性。

```
[NotMapped]
public string BlogCode
{
    get
    {
        return Title.Substring(0, 1) + ":" + BloggerName.Substring(0, 1);
    }
}
```

ComplexType

请不要在一组类中描述域实体，然后将这些类分层以描述完整的实体。例如，可以将名为 BlogDetails 的类添加到模型。

```
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

请注意，BlogDetails 没有任何类型的键属性。在域驱动设计中，BlogDetails 称为值对象。实体框架将值对象引用为复杂类型。不能自行跟踪复杂类型。

但作为博客类中的属性，BlogDetails 将作为博客对象的一部分进行跟踪。为了使代码优先识别这一点，必须将 BlogDetails 类标记为 ComplexType。

```
[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

    [MaxLength(250)]
    public string Description { get; set; }
}
```

现在，可以在博客类中添加一个属性，用于表示该博客的 BlogDetails。

```
public BlogDetails BlogDetail { get; set; }
```

在数据库中，博客表将包含博客的所有属性，包括其 BlogDetail 属性中包含的属性。默认情况下，每个名称的前面都有复杂类型 BlogDetail 的名称。

dbo.Blogs
└─ Columns
└─ PrimaryTrackingKey (PK, int, not null)
└─ Title (nvarchar(128), not null)
└─ BloggerName (nvarchar(10), null)
└─ BlogDetail_DateCreated (datetime, null)
└─ BlogDetail_Description (nvarchar(250), null)

ConcurrencyCheck

ConcurrencyCheck 批注允许标记一个或多个属性，用户在编辑或删除实体时，此属性将用于数据库中的并发检查。如果你使用的是 EF 设计器，则会将属性的 ConcurrencyMode 设置为 Fixed。

让我们通过将其添加到 BloggerName 属性来了解 ConcurrencyCheck 的工作原理。

```
[ConcurrencyCheck, MaxLength(10, ErrorMessage="BloggerName must be 10 characters or less"),MinLength(5)]
public string BloggerName { get; set; }
```

当调用 SaveChanges 时, 由于 BloggerName 字段上的 ConcurrencyCheck 批注, 将在更新中使用该属性的原始值。此命令将尝试通过筛选键值而不是 BloggerName 的原始值来查找正确的行。下面是发送到数据库的 UPDATE 命令的关键部分, 您可以在其中看到命令将更新 PrimaryTrackingKey 为 1 的行, BloggerName 为 "Julie", 这是从数据库中检索到该博客时的原始值。

```
where (([PrimaryTrackingKey] = @4) and ([BloggerName] = @5))
@4=1,@5=N'Julie'
```

如果用户同时更改了博客的博客名称, 则此更新将会失败, 并且你将收到需要处理的 DbUpdateConcurrencyException。

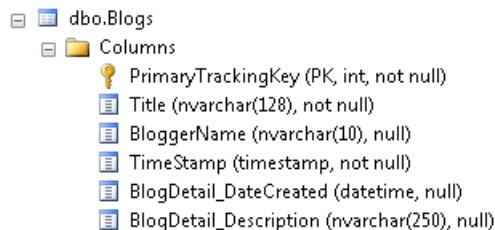
TimeStamp

更常见的情况是使用 rowversion 或时间戳字段进行并发检查。而不是使用 ConcurrencyCheck 批注, 只要属性的类型是字节数组, 就可以使用更具体的时间戳注释。Code first 会将 Timestamp 属性视为与 ConcurrencyCheck 属性相同, 但它还将确保代码第一次生成的数据库字段不可为 null。给定的类中只能有一个时间戳属性。

将以下属性添加到博客类:

```
[Timestamp]
public Byte[] TimeStamp { get; set; }
```

在数据库表中生成一个不可以为 null 的时间戳列的代码。



dbo.Blogs
Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), not null)
BloggerName (nvarchar(10), null)
TimeStamp (timestamp, not null)
BlogDetail_DateCreated (datetime, null)
BlogDetail_Description (nvarchar(250), null)

表和列

如果允许 Code First 创建数据库, 则可能需要更改要创建的表和列的名称。您还可以将 Code First 与现有数据库一起使用。但并非总是域中类和属性的名称与数据库中的表和列的名称相匹配。

我的类称为博客和惯例, 代码优先假设这将映射到名为 Blog 的表。如果不是这种情况, 则可以指定具有表属性的表的名称。例如, 批注正在将表名指定为 InternalBlogs。

```
[Table("InternalBlogs")]
public class Blog
```

在指定映射列的特性时, 列批注更熟练地使用。您可以规定名称、数据类型, 甚至是列在表中的显示顺序。下面是列属性的示例。

```
[Column("BlogDescription", TypeName="ntext")]
public String Description {get;set;}
```

不要将列的 TypeName 特性与 DataType DataAnnotation 混淆。数据类型是一种用于 UI 的批注, Code First 会将其忽略。

下面是重新生成表后的表。表名称已更改为 InternalBlogs, 并且来自复杂类型的说明列现在为 BlogDescription。由于在注释中指定了名称, 因此 code first 将不会使用以复杂类型名称开头的列名称。

```
dbo.InternalBlogs
Columns
PrimaryTrackingKey (PK, int, not null)
Title (nvarchar(128), not null)
BloggerName (nvarchar(10), null)
TimeStamp (timestamp, not null)
BlogDetail_DateCreated (datetime, null)
BlogDescription (ntext, null)
```

DatabaseGenerated

重要的数据库功能是具有计算属性的能力。如果要将 Code First 类映射到包含计算列的表, 则不希望实体框架尝试更新这些列。但是, 在插入或更新数据后, 您确实需要 EF 从数据库返回这些值。可以使用 DatabaseGenerated 批注来标记类中的这些属性以及计算所得的枚举。其他枚举为 None 和 Identity。

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public DateTime DateCreated { get; set; }
```

当代码优先生成数据库时, 可以使用在字节或时间戳列上生成的数据库, 否则, 只应在指向现有数据库时使用此数据库, 因为 code first 无法确定计算列的公式。

如上所述, 在默认情况下, 作为整数的键属性将成为数据库中的标识键。这与将 DatabaseGenerated 设置为 DatabaseGeneratedOption 相同。如果您不希望它是标识密钥, 则可以将该值设置为 DatabaseGeneratedOption。

Index

NOTE

Ef 6.1 ■往上-索引属性是在实体框架6.1 中引入的。如果你使用的是早期版本, 则本部分中的信息不适用。

您可以使用IndexAttribute对一个或多个列创建索引。将属性添加到一个或多个属性时, 将导致 EF 在创建数据库时在数据库中创建相应的索引, 或者如果使用 Code First 迁移, 则为相应的CreateIndex调用基架。

例如, 下面的代码将生成一个索引, 该索引是在数据库的 "发布" 表的 "分级" 列上创建的。

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    [Index]
    public int Rating { get; set; }
    public int BlogId { get; set; }
}
```

默认情况下，索引将命名为IX_<属性名称>（在上面的示例中为 IX_评级）。您还可以指定索引的名称。下面的示例指定索引应命名为PostRatingIndex。

```
[Index("PostRatingIndex")]
public int Rating { get; set; }
```

默认情况下，索引是不唯一的，但您可以使用IsUnique命名参数指定索引应是唯一的。下面的示例对用户的登录名引入唯一索引。

```
public class User
{
    public int UserId { get; set; }

    [Index(IsUnique = true)]
    [StringLength(200)]
    public string Username { get; set; }

    public string DisplayName { get; set; }
}
```

多列索引

跨多个列的索引通过在给定表的多个索引批注中使用相同的名称来指定。创建多列索引时，需要指定索引中列的顺序。例如，下面的代码创建一个名为IX_BlogIdAndRating的分级和BlogId的多列索引。BlogId是索引中的第一列，而评级是第二列。

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    [Index("IX_BlogIdAndRating", 2)]
    public int Rating { get; set; }
    [Index("IX_BlogIdAndRating", 1)]
    public int BlogId { get; set; }
}
```

关系属性：InverseProperty 和 ForeignKey

NOTE

本页提供有关使用数据批注设置 Code First 模型中的关系的信息。有关 EF 中的关系以及如何使用关系访问和操作数据的一般信息，请参阅[关系 & 导航属性](#)。*

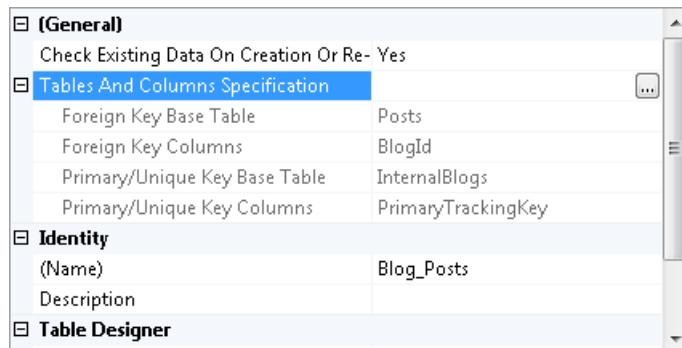
Code first 约定将负责您的模型中最常见的关系，但在某些情况下，需要帮助。

更改博客类中键属性的名称时，会创建一个与发布的关系相关的问题。

生成数据库时，代码优先会在 Post 类中看到 BlogId 属性，并将其识别为与博客类的外键匹配的类名和 "Id"。但博客类中没有 BlogId 属性。此解决方案的作用是在 Post 中创建一个导航属性，并使用外 DataAnnotation 帮助代码首先了解如何使用 BlogId 属性构建这两个类之间的关系，以及如何在数据。

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    [ForeignKey("BlogId")]
    public Blog Blog { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

数据库中的约束显示 InternalBlogs 和 BlogId 之间的关系。



当类之间存在多个关系时，将使用 InverseProperty。

在 Post 类中，你可能需要跟踪博客文章的作者，以及编辑者。下面是 Post 类的两个新导航属性。

```
public Person CreatedBy { get; set; }
public Person UpdatedBy { get; set; }
```

还需要添加到这些属性引用的 Person 类中。Person 类将导航属性返回到张贴内容，一个用于该用户撰写的所有帖子，另一个用于该人员所更新的所有帖子。

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Post> PostsWritten { get; set; }
    public List<Post> PostsUpdated { get; set; }
}
```

Code first 不能自行匹配两个类中的属性。用于发布的数据库表应具有 System.createdby 人员的一个外键，另一个用于 UpdatedBy 人员，而 code first 将创建四个外键属性：Person_Id、Person_Id1、System.createdby_Id 和 UpdatedBy_Id。

		dbo.Posts
		Columns
		Id (PK, int, not null)
		Title (nvarchar(128), null)
		DateCreated (datetime, not null)
		Content (nvarchar(128), null)
		BlogId (FK, int, not null)
		Person_Id (FK, int, null)
		Person_Id1 (FK, int, null)
		CreatedBy_Id (FK, int, null)
		UpdatedBy_Id (FK, int, null)

若要解决这些问题，可以使用 `InverseProperty` 批注来指定属性的对齐方式。

```
[InverseProperty("CreatedBy")]
public List<Post> PostsWritten { get; set; }

[InverseProperty("UpdatedBy")]
public List<Post> PostsUpdated { get; set; }
```

因为 Person 中的 `PostsWritten` 属性知道这指的是 Post 类型，它将生成与 `System.createdby` 的关系。同样，`PostsUpdated` 将连接到 `UpdatedBy`。和 code first 不会创建额外的外键。

		dbo.Posts
		Columns
		Id (PK, int, not null)
		Title (nvarchar(128), null)
		DateCreated (datetime, not null)
		Content (nvarchar(128), null)
		BlogId (FK, int, not null)
		CreatedBy_Id (FK, int, null)
		UpdatedBy_Id (FK, int, null)

摘要

DataAnnotations 不仅使你能够在代码优先类中描述客户端和服务器端验证，还允许你增强，甚至更正代码优先根据其约定对类进行的假设。使用 DataAnnotations，不仅可以驱动数据库架构生成，还可以将代码的第一类映射到预先存在的数据库。

尽管它们非常灵活，但请记住，DataAnnotations 仅提供您可以在代码优先类上进行的最常见的配置更改。若要为某些边缘事例配置类，应查看备用配置机制，Code First 的 "流畅" API。

定义 DbSet

2020/3/11 ·

使用 Code First 工作流进行开发时，可定义一个派生 DbContext，用于表示与数据库的会话，并为模型中的每个类型公开一个 DbSet。本主题介绍可用于定义 DbSet 属性的各种方式。

具有 DbSet 属性的 DbContext

Code First 示例中所示的常见情况是，对于模型的实体类型，DbContext 具有公共自动 DbSet 属性。例如：

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

当在 Code First 模式下使用时，这将博客和公告配置为实体类型，并配置可从此访问的其他类型。此外，DbContext 会自动为每个属性调用 setter，以设置相应 DbSet 的实例。

具有 IDbSet 属性的 DbContext

在某些情况下，例如创建模拟或 fakes 时，使用接口来声明集属性更有用。在这种情况下，可以使用 IDbSet 接口代替 DbSet。例如：

```
public class BloggingContext : DbContext
{
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
}
```

此上下文的工作方式与使用 DbSet 类作为其设置属性的上下文完全相同。

具有只读 set 属性的 DbContext

如果你不希望为 DbSet 或 IDbSet 属性公开公共资源库，则可以改为创建只读属性并自行创建集实例。例如：

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs
    {
        get { return Set<Blog>(); }
    }

    public DbSet<Post> Posts
    {
        get { return Set<Post>(); }
    }
}
```

请注意，DbContext 将缓存从 Set 方法返回的 DbSet 的实例，使每个属性在每次调用时都返回相同的实例。

Code First 的实体类型的发现的工作方式与使用公共 getter 和 setter 的属性相同。

枚举支持 -Code First

2020/3/12 •

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

此视频和分步演练演示了如何使用实体框架 Code First 的枚举类型。它还演示了如何在 LINQ 查询中使用枚举。

本演练将使用 Code First 创建新数据库，但也可以使用[Code First 映射到现有数据库](#)。

实体框架5中引入了枚举支持。若要使用枚举、空间数据类型和表值函数等新功能，则必须以 .NET Framework 4.5 为目标。默认情况下，Visual Studio 2012 面向 .NET 4.5。

在实体框架中，枚举可以具有以下基础类型：`Byte`、`Int16`、`Int32`、`Int64`或`SByte`。

观看视频

此视频演示如何使用实体框架 Code First 的枚举类型。它还演示了如何在 LINQ 查询中使用枚举。

主讲人 :Julia Kornich

视频 :WMV | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

你将需要安装 Visual Studio 2012、旗舰版、高级版、专业版或 Web Express edition 才能完成此演练。

设置项目

1. 打开 Visual Studio 2012
2. 在 "文件" 菜单上，指向 "新建"，然后单击 "项目"
3. 在左窗格中，单击 "Visual C#"，然后选择控制台模板
4. 输入EnumCodeFirst作为项目名称，然后单击 "确定"

使用 Code First 定义新模型

使用 Code First 开发时，通常首先编写定义概念(域)模型 .NET Framework 类。下面的代码定义了部门类。

该代码还定义了 DepartmentNames 枚举。默认情况下，枚举的类型为`int`。部门类的 Name 属性为 DepartmentNames 类型。

打开 Program.cs 文件并粘贴以下类定义。

```
public enum DepartmentNames
{
    English,
    Math,
    Economics
}

public partial class Department
{
    public int DepartmentID { get; set; }
    public DepartmentNames Name { get; set; }
    public decimal Budget { get; set; }
}
```

定义 DbContext 派生类型

除了定义实体外，还需要定义派生自 DbContext 的类，并公开 DbSet< TEntity > 属性。 DbSet< TEntity > 属性使上下文知道要包括在模型中的类型。

DbContext 派生类型的实例在运行时管理实体对象，这包括使用数据库中的数据填充对象、更改跟踪以及将数据保存到数据库。

DbContext 和 DbSet 类型是在 EntityFramework 程序集中定义的。我们将使用 EntityFramework NuGet 包添加对此 DLL 的引用。

1. 在解决方案资源管理器中，右键单击项目名称。
2. 选择 "管理 NuGet 包..."
3. 在 "管理 NuGet 包" 对话框中，选择 "联机" 选项卡，然后选择 "EntityFramework" 包。
4. 单击"安装"

请注意，除 EntityFramework 程序集之外，还添加了对 System.componentmodel 和 DataAnnotations 程序集的引用。

在 Program.cs 文件的顶部，添加以下 using 语句：

```
using System.Data.Entity;
```

在 Program.cs 中，添加上下文定义。

```
public partial class EnumTestContext : DbContext
{
    public DbSet<Department> Departments { get; set; }
}
```

保留和检索数据

打开 Program.cs 文件，其中定义了 Main 方法。将以下代码添加到 Main 函数中。该代码将新的部门对象添加到上下文中。然后，它会保存数据。此代码还执行 LINQ 查询，该查询返回名称为 DepartmentNames 的部门。

```
using (var context = new EnumTestContext())
{
    context.Departments.Add(new Department { Name = DepartmentNames.English });

    context.SaveChanges();

    var department = (from d in context.Departments
                      where d.Name == DepartmentNames.English
                      select d).FirstOrDefault();

    Console.WriteLine(
        "DepartmentID: {0} Name: {1}",
        department.DepartmentID,
        department.Name);
}
```

编译并运行该应用程序。该程序生成以下输出：

```
DepartmentID: 1 Name: English
```

查看生成的数据库

首次运行应用程序时，实体框架会为您创建一个数据库。由于我们安装了 Visual Studio 2012，因此将在 LocalDB 实例上创建数据库。默认情况下，实体框架在派生上下文的完全限定名称后命名数据库（在此示例中为 `EnumTestContext`）。随后将使用现有数据库的时间。

请注意，如果在创建数据库后对模型进行了任何更改，则应使用 Code First 迁移来更新数据库架构。有关使用迁移的示例，请参阅[Code First 到新的数据库](#)。

若要查看数据库和数据，请执行以下操作：

1. 在 Visual Studio 2012 主菜单中，选择“查看 -> SQL Server 对象资源管理器”。
2. 如果 LocalDB 不在服务器列表中，请在 SQL Server 上单击鼠标右键按钮，然后选择“添加 SQL Server 使用默认 Windows 身份验证连接到 LocalDB 实例”
3. 展开 LocalDB 节点
4. 展开“数据库”文件夹以查看新数据库并浏览到“部门”表，该 Code First 不会创建映射到枚举类型的表
5. 若要查看数据，请右键单击该表，然后选择“查看数据”

Summary

在本演练中，我们介绍了如何使用实体框架 Code First 的枚举类型。

空间 Code First

2020/3/12 •

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

视频和分步演练演示了如何使用实体框架 Code First 映射空间类型。它还演示了如何使用 LINQ 查询查找两个位置之间的距离。

本演练将使用 Code First 创建新数据库，但也可以使用[Code First 到现有数据库](#)。

实体框架5中引入了空间类型支持。请注意，若要使用空间类型、枚举和表值函数等新功能，则必须以 .NET Framework 4.5 为目标。默认情况下，Visual Studio 2012 面向 .NET 4.5。

若要使用空间数据类型，还必须使用具有空间支持的实体框架提供程序。有关详细信息，请参阅[提供程序对空间类型的支持](#)。

主要的空间数据类型有两种：地理和几何。Geography 数据类型存储椭圆体的数据（例如 GPS 纬度和经度坐标）。Geometry 数据类型表示欧氏（平面）坐标系。

观看视频

此视频演示如何实体框架 Code First 映射空间类型。它还演示了如何使用 LINQ 查询查找两个位置之间的距离。

主讲人 :Julia Kornich

视频 :WMV | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

你将需要安装 Visual Studio 2012、旗舰版、高级版、专业版或 Web Express edition 才能完成此演练。

设置项目

1. 打开 Visual Studio 2012
2. 在“文件”菜单上，指向“新建”，然后单击“项目”
3. 在左窗格中，单击“Visual C#”，然后选择控制台模板
4. 输入SpatialCodeFirst作为项目名称，然后单击“确定”

使用 Code First 定义新模型

使用 Code First 开发时，通常首先编写定义概念（域）模型 .NET Framework 类。下面的代码定义了大学类。

该大学具有 DbGeography 类型的 Location 属性。若要使用 DbGeography 类型，必须添加对 System.web 程序集的引用，并使用语句添加 node.js。

打开 Program.cs 文件，并将以下 using 语句粘贴到文件顶部：

```
using System.Data.Spatial;
```

将以下大学类定义添加到 Program.cs 文件。

```
public class University
{
    public int UniversityID { get; set; }
    public string Name { get; set; }
    public DbGeography Location { get; set; }
}
```

定义 DbContext 派生类型

除了定义实体外，还需要定义派生自 DbContext 的类，并公开 DbSet< TEntity > 属性。 DbSet< TEntity > 属性使上下文知道要包括在模型中的类型。

DbContext 派生类型的实例在运行时管理实体对象，这包括使用数据库中的数据填充对象、更改跟踪以及将数据保存到数据库。

DbContext 和 DbSet 类型是在 EntityFramework 程序集中定义的。我们将使用 EntityFramework NuGet 包添加对此 DLL 的引用。

1. 在解决方案资源管理器中，右键单击项目名称。
2. 选择 "管理 NuGet 包 ..."。
3. 在 "管理 NuGet 包" 对话框中，选择 "联机" 选项卡，然后选择 "EntityFramework" 包。
4. 单击"安装"

请注意，除 EntityFramework 程序集之外，还添加了对 System.ComponentModel.DataAnnotations 程序集的引用。

在 Program.cs 文件的顶部，添加以下 using 语句：

```
using System.Data.Entity;
```

在 Program.cs 中，添加上下文定义。

```
public partial class UniversityContext : DbContext
{
    public DbSet<University> Universities { get; set; }
}
```

保留和检索数据

打开 Program.cs 文件，其中定义了 Main 方法。将以下代码添加到 Main 函数中。

该代码将两个新的大学对象添加到上下文中。空间属性使用 DbGeography.FromText 方法进行初始化。将 WellKnownText 表示的地理点传递给方法。然后，该代码将保存数据。然后，将构造并执行 LINQ 查询，该查询返回其位置与指定位置最接近的大学对象。

```

using (var context = new UniversityContext ())
{
    context.Universities.Add(new University()
    {
        Name = "Graphic Design Institute",
        Location = DbGeography.FromText("POINT(-122.336106 47.605049)"),
    });

    context.Universities.Add(new University()
    {
        Name = "School of Fine Art",
        Location = DbGeography.FromText("POINT(-122.335197 47.646711)"),
    });

    context.SaveChanges();

    var myLocation = DbGeography.FromText("POINT(-122.296623 47.640405)");

    var university = (from u in context.Universities
                      orderby u.Location.Distance(myLocation)
                      select u).FirstOrDefault();

    Console.WriteLine(
        "The closest University to you is: {0}.",
        university.Name);
}

```

编译并运行该应用程序。该程序生成以下输出：

```
The closest University to you is: School of Fine Art.
```

查看生成的数据库

首次运行应用程序时，实体框架会为您创建一个数据库。由于我们安装了 Visual Studio 2012，因此将在 LocalDB 实例上创建数据库。默认情况下，实体框架在派生上下文的完全限定名称（在此示例中为 SpatialCodeFirst. UniversityContext）之后对数据库进行命名。随后将使用现有数据库的时间。

请注意，如果在创建数据库后对模型进行了任何更改，则应使用 Code First 迁移来更新数据库架构。有关使用迁移的示例，请参阅[Code First 到新的数据库](#)。

若要查看数据库和数据，请执行以下操作：

1. 在 Visual Studio 2012 主菜单中，选择“查看 -> SQL Server 对象资源管理器”。
2. 如果 LocalDB 不在服务器列表中，请在 SQL Server 上单击鼠标右键按钮，然后选择“添加 SQL Server 使用默认 Windows 身份验证连接到 LocalDB 实例”
3. 展开 LocalDB 节点
4. 展开“数据库”文件夹，以查看新数据库并浏览到大学表
5. 若要查看数据，请右键单击该表，然后选择“查看数据”

Summary

在本演练中，我们介绍了如何在实体框架 Code First 中使用空间类型。

Code First 约定

2020/3/11 •

Code First 使你可以使用C#或 Visual Basic .net 类来描述模型。使用约定检测模型的基本形状。约定是一组规则，这些规则用于在使用 Code First 时，基于类定义自动配置概念模型。约定是在 ModelConfiguration 命名空间中定义的。

您可以通过使用数据批注或 Fluent API 来进一步配置模型。优先级是通过 Fluent API 后跟数据注释和约定的方式进行配置。有关详细信息，请参阅[数据批注、流畅的 api 关系、熟知的 api 类型 & 属性](#)和[VB.NET 的流畅 api](#)。

[API 文档](#)中提供了 Code First 约定的详细列表。本主题概述了 Code First 使用的约定。

类型发现

使用 Code First 开发时，通常首先编写定义概念(域)模型 .NET Framework 类。除了定义类之外，还需要让`DbContext`知道要在模型中包含的类型。为此，您定义了一个从`DbContext`派生的上下文类，并公开了要成为模型一部分的类型的`DbSet`属性。Code First 将包含这些类型，并且还将请求任何引用的类型，即使引用的类型在不同的程序集中定义也是如此。

如果你的类型参与了继承层次结构，则为基类定义`DbSet`属性就足够了，如果派生类型与基类位于同一程序集中，则它们将自动包含。

在下面的示例中，在`SchoolEntities`类(部门)上只定义了一个`DbSet`属性。Code First 使用此属性来发现和请求任何引用的类型。

```

public class SchoolEntities : DbContext
{
    public DbSet<Department> Departments { get; set; }
}

public class Department
{
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public string Location { get; set; }
    public string Days { get; set; }
    public System.DateTime Time { get; set; }
}

```

如果要从模型中排除某一类型，请使用`NotMapped`属性或`DbModelBuilder` Fluent API。

```
modelBuilder.Ignore<Department>();
```

Primary Key 约定

Code First 推断，如果类的属性命名为 "ID"（不区分大小写）或类名称后跟 "ID"，则属性是主键。如果主键属性的类型是数值或 GUID，则将其配置为标识列。

```

public class Department
{
    // Primary key
    public int DepartmentID { get; set; }

    ...
}
```

关系约定

在实体框架中，导航属性提供了一种方法，用于在两个实体类型之间导航关系。针对对象参与到其中的每个关系，各对象均可以具有导航属性。使用导航属性，您可以在两个方向上导航和管理关系，返回引用对象（如果重数为1或零）或集合（如果重数为多个）。Code First 根据在您的类型上定义的导航属性推断关系。

除了导航属性外，我们还建议你在表示依赖对象的类型上包含外键属性。任何数据类型与主体主键属性相同且名称遵循以下格式之一的任何属性都表示关系的外键：“<导航属性名称><主体主键属性名称>”、“<主体类名称><primary key 属性名称>”或“<主体主键属性名称>”。如果找到多个匹配项，则优先顺序按上面列出的顺序提供。外键检测不区分大小写。当检测到外键属性时，Code First 根据外键的为 null 性推断关系的重数。如果该属性可以为 null，则会将该关系注册为可选；否则，将根据需要注册此关系。

如果从属实体上的外键不可为 null，则 Code First 在关系上设置级联删除。如果从属实体上的外键可为 null，则 Code First 不会对关系设置级联删除，并且在删除主体时，外键将设置为 null。可以使用 Fluent API 重写约定检测到的重数和级联删除行为。

在下面的示例中，导航属性和外键用于定义部门和课程类之间的关系。

```
public class Department
{
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
}
```

NOTE

如果相同类型之间有多个关系（例如，假设你定义了 `person` 和 `book` 类，其中 `Person` 类包含 `ReviewedBooks` 和 `AuthoredBooks` 导航属性，`Book` 类包含 `Author` 和 ■ 导航属性），则需要通过使用数据批注或 Fluent API 来手动配置关系。有关详细信息，请参阅 [数据批注-关系](#) 和 [熟知 API 关系](#)。

复杂类型约定

如果 Code First 发现不能推断出主键的类定义，且没有通过数据批注或 Fluent API 注册主键，则该类型将自动注册为复杂类型。复杂类型检测还要求类型没有引用实体类型的属性，并且没有从另一类型的集合属性中引用。给定以下类定义 Code First 将推断，[详细信息](#) 是复杂类型，因为它没有主键。

```
public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}
```

连接字符串约定

若要了解 DbContext 用于发现连接所使用的约定, 请参阅[连接和模型](#)。

删除约定

您可以删除在 ModelConfiguration 命名空间中定义的任何约定。下面的示例将删除 PluralizingTableNameConvention。

```
public class SchoolEntities : DbContext
{
    . .

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention, the generated tables
        // will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

自定义约定

EF6 中支持自定义约定。有关详细信息, 请参阅[自定义 Code First 约定](#)。

自定义 Code First 约定

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

使用时 Code First 模型将使用一组约定从类进行计算。默认[Code First 约定](#)确定哪些属性将成为实体的主键、实体映射到的表的名称，以及在默认情况下，小数列的精度和小数位数。

有时，这些默认约定并不适用于您的模型，并且您必须通过使用数据注释或流畅的 API 来配置多个单独的实体来解决它们。自定义 Code First 约定允许您定义自己的约定，为模型提供配置默认值。在本演练中，我们将探讨不同类型的自定义约定以及如何创建它们。

基于模型的约定

本页介绍用于自定义约定的 `DbModelBuilder` API。此 API 应该足以用于创作大多数自定义约定。但是，还可以创作基于模型的约定-在创建最终模型后对其进行操作的约定-用于处理高级方案。有关详细信息，请参阅[基于模型的约定](#)。

我们的模型

首先，我们定义一个可用于约定的简单模型。将以下类添加到项目。

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;

public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ProductContext>());
    }

    public DbSet<Product> Products { get; set; }
}

public class Product
{
    public int Key { get; set; }
    public string Name { get; set; }
    public decimal? Price { get; set; }
    public DateTime? ReleaseDate { get; set; }
    public ProductCategory Category { get; set; }
}

public class ProductCategory
{
    public int Key { get; set; }
    public string Name { get; set; }
    public List<Product> Products { get; set; }
}

```

自定义约定简介

我们来编写一种约定，将名为 Key 的任何属性配置为其实体类型的主键。

在模型生成器上启用约定，可以通过在上下文中重写 OnModelCreating 来访问这些约定。更新 ProductContext 类，如下所示：

```

public class ProductContext : DbContext
{
    static ProductContext()
    {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<ProductContext>());
    }

    public DbSet<Product> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Properties()
            .Where(p => p.Name == "Key")
            .Configure(p => p.IsKey());
    }
}

```

现在，模型中名为 Key 的任何属性都将配置为其一部分的任何实体的主键。

还可以通过筛选要配置的属性类型，使约定更加具体：

```
modelBuilder.Properties<int>()
    .Where(p => p.Name == "Key")
    .Configure(p => p.IsKey());
```

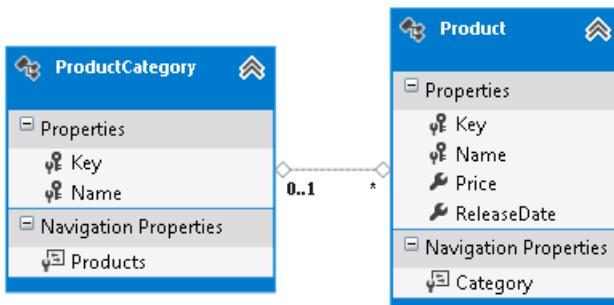
这会将名为 Key 的所有属性配置为其实体的主键，但前提是它们是整数。

IsKey 方法的一个有趣功能是它是附加的。这意味着，如果在多个属性上调用 IsKey，它们都将成为复合键的一部分。需要注意的一点是，如果为某个键指定了多个属性，还必须为这些属性指定顺序。为此，可以调用 HasColumnOrder 方法，如下所示：

```
modelBuilder.Properties<int>()
    .Where(x => x.Name == "Key")
    .Configure(x => x.IsKey().HasColumnOrder(1));

modelBuilder.Properties()
    .Where(x => x.Name == "Name")
    .Configure(x => x.IsKey().HasColumnOrder(2));
```

此代码将配置模型中的类型，使其包含 int 键列和字符串名称列组成的组合键。如果在设计器中查看模型，它将如下所示：



属性约定的另一个示例是将我的模型中的所有日期时间属性配置为映射到 SQL Server 中的 datetime2 类型，而不是 DateTime。可以通过以下方式实现此目的：

```
modelBuilder.Properties<DateTime>()
    .Configure(c => c.HasColumnType("datetime2"));
```

约定类

定义约定的另一种方法是使用约定类封装约定。使用约定类时，将创建一个从 ModelConfiguration 命名空间中的约定类继承的类型。

可以通过执行以下操作，创建一个具有前文约定的约定类：

```
public class DateTime2Convention : Convention
{
    public DateTime2Convention()
    {
        this.Properties<DateTime>()
            .Configure(c => c.HasColumnType("datetime2"));
    }
}
```

若要告诉 EF 使用此约定，请将其添加到 OnModelCreating 中的约定集合，如果已执行以下操作，演练将如下所

示：

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Properties<int>()
        .Where(p => p.Name.EndsWith("Key"))
        .Configure(p => p.IsKey());

    modelBuilder.Conventions.Add(new DateTime2Convention());
}
```

如您所见，我们将约定的实例添加到约定集合。从约定继承提供了一种在团队或项目之间进行分组和共享约定的便利方法。例如，你可以有一个类库，其中包含所有组织项目使用的一组通用约定。

自定义特性

约定的另一种很好的用途是启用在配置模型时使用的新属性。为了说明这一点，让我们创建一个属性，用于将字符串属性标记为非 Unicode。

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
public class NonUnicode : Attribute
{}
```

现在，让我们创建一个约定，以将此属性应用于模型：

```
modelBuilder.Properties()
    .Where(x => x.GetCustomAttributes(false).OfType<NonUnicode>().Any())
    .Configure(c => cIsUnicode(false));
```

通过此约定，我们可以将 NonUnicode 属性添加到字符串的任何属性，这意味着数据库中的列将存储为 varchar 而不是 nvarchar。

关于此约定，需要注意的一点是，如果将 NonUnicode 属性放在字符串属性以外的任何内容中，则会引发异常。这是因为不能在字符串以外的任何类型上配置IsUnicode。如果发生这种情况，则可以使约定更为具体，使其筛选掉不是字符串的任何内容。

尽管上述约定适用于定义自定义属性，但还有另一个 API 可以更容易地使用，尤其是在你想要使用属性类中的属性时。

在此示例中，我们将更新属性，并将其更改为IsUnicode 属性，如下所示：

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)]
internal classIsUnicode : Attribute
{
    public bool Unicode { get; set; }

    publicIsUnicode(bool isUnicode)
    {
        Unicode = isUnicode;
    }
}
```

完成此设置后，可以在属性上设置一个布尔值，以指示该属性是否应为 Unicode。为此，我们可以通过访问配置类的 ClrProperty 来执行此操作，如下所示：

```
modelBuilder.Properties()
    .Where(x => x.GetCustomAttributes(false).OfType<IsUnicode>().Any())
    .Configure(c => cIsUnicode(c.Clr PropertyInfo.GetCustomAttribute<IsUnicode>().Unicode));
```

这非常简单，但通过使用约定 API 的 Having 方法，可以更简洁地实现此方法。Having 方法具有类型为 Func< PropertyInfo, T > 的参数，该参数接受与 Where 方法相同的 PropertyInfo，但预期返回对象。如果返回的对象为 null，则不会配置属性，这意味着，你可以像在中那样筛选 out 属性，但不同之处在于它还将捕获返回的对象并将其传递给 Configure 方法。这类似于以下内容：

```
modelBuilder.Properties()
    .Having(x =>x.GetCustomAttributes(false).OfType<IsUnicode>().FirstOrDefault())
    .Configure((config, att) => configIsUnicode(att.Unicode));
```

自定义特性并不是使用 Having 方法的唯一理由，在配置类型或属性时，您需要对其进行筛选的任何位置都很有用。

配置类型

到目前为止，我们的所有约定都是关于属性的，但也有另一个用于配置模型中的类型的约定 API 区域。经验类似于我们目前所见到的约定，但配置内的选项将位于实体而不是属性级别。

类型级别约定的一项功能对于更改表命名约定非常有用，它可以映射到与 EF 默认值不同的现有架构，也可以使用不同的命名约定来创建新的数据库。为此，我们首先需要一个方法，该方法可以接受模型中类型的 TypeInfo，并返回该类型的表名称应为：

```
private string GetTableName(Type type)
{
    var result = Regex.Replace(type.Name, ".[A-Z]", m => m.Value[0] + "_" + m.Value[1]);

    return result.ToLower();
}
```

此方法采用类型并返回一个字符串，该字符串使用带下划线的小写形式而不是 CamelCase。在我们的模型中，这意味着 ProductCategory 类将映射到名为 product_category 的表，而不是 ProductCategories。

获得该方法后，我们可以在此类约定中调用它：

```
modelBuilder.Types()
    .Configure(c => c.ToTable(GetTableName(c.ClrType)));
```

此约定将模型中的每个类型配置为映射到 GetTableName 方法返回的表名。此约定等效于使用熟知 API 为模型中的每个实体调用 ToTable 方法。

需要注意的一点是，当你调用 ToTable EF 时，将采用你提供的字符串作为确切的表名称，而不是在确定表名称时通常会执行的任何复数形式。这就是我们约定中的表名是 product_类别而不是产品_类别的原因。我们可以通过调用复数形式服务自己的约定来解决这一问题。

在下面的代码中，我们将使用 EF6 中添加的[依赖项解析](#)功能检索 EF 使用的复数形式服务并复数形式我们的表名。

```

private string GetTableName(Type type)
{
    var pluralizationService = DbConfiguration.DependencyResolver.GetService<IPluralizationService>();

    var result = pluralizationService.Pluralize(type.Name);

    result = Regex.Replace(result, ".[A-Z]", m => m.Value[0] + "_" + m.Value[1]);

    return result.ToLower();
}

```

NOTE

GetService 的泛型版本是 DependencyResolution 命名空间中的扩展方法，需要将 using 语句添加到你的上下文中才能使用该语句。

ToTable 和继承

ToTable 的另一个重要方面是，如果您将某一类型显式映射到给定的表，则可以更改 EF 将使用的映射策略。如果为继承层次结构中的每个类型调用 ToTable，并将类型名称作为表的名称传递，则会将默认的每个层次结构一个表(TPH)映射策略更改为每种类型一个表(TPT)。描述这一点的最佳方式是 with 一个具体的示例：

```

public class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Manager : Employee
{
    public string SectionManaged { get; set; }
}

```

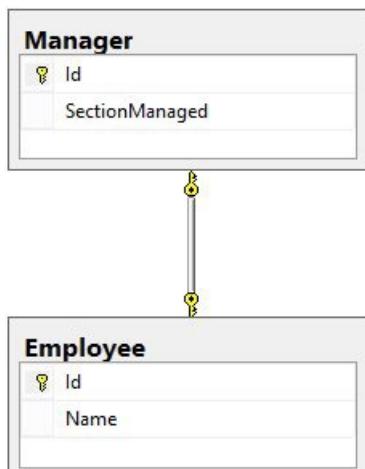
默认情况下，员工和经理都映射到数据库中的同一表(员工)。该表将包含具有鉴别器列的员工和经理，该列将告诉您每一行中存储的实例类型。这是 TPH 映射，因为层次结构中有一个表。但是，如果在这两个类上调用 ToTable，则每个类型都将映射到其自己的表，也称为 TPT，因为每个类型都有其自己的表。

```

modelBuilder.Types()
    .Configure(c=>c.ToTable(c.ClrType.Name));

```

上面的代码将映射到如下所示的表结构：



可以通过以下几种方式来避免这种情况并维护默认的 TPH 映射：

1. 为层次结构中的每个类型调用具有相同表名称的 ToTable。
2. 仅在层次结构的基类(在本示例中为 employee)上调用 ToTable。

执行顺序

约定的运行方式与精通 API 相同。这意味着，如果您编写两个约定来配置同一属性的相同选项，最后一个规则执行 wins。例如，在下面的代码中，所有字符串的最大长度都设置为500，但然后，在模型中将名为 Name 的所有属性都配置为具有最大长度250。

```
modelBuilder.Properties<string>()
    .Configure(c => c.HasMaxLength(500));

modelBuilder.Properties<string>()
    .Where(x => x.Name == "Name")
    .Configure(c => c.HasMaxLength(250));
```

由于将最大长度设置为250的约定是在将所有字符串设置为500后，因此模型中名为 "名称" 的所有属性的 MaxLength 都为250，而任何其他字符串(如 "说明")都是500。以这种方式使用约定意味着你可以为模型中的类型或属性提供一般约定，然后将其覆盖为不同的子集。

在特定情况下，还可以使用熟知的 API 和数据批注来替代约定。在上面的示例中，如果我们使用了 "熟知 API" 设置属性的最大长度，则可以将其放在约定之前或之后，因为更具体的熟知 API 将通过更常规的配置约定。

内置约定

由于自定义约定可能受默认 Code First 约定的影响，因此添加约定以在其他约定之前或之后运行可能会很有用。为此，可以在派生的 DbContext 上使用约定集合的 AddBefore 和 AddAfter 方法。下面的代码将添加前面创建的约定类，使其在内置密钥发现约定之前运行。

```
modelBuilder.Conventions.AddBefore<IdKeyDiscoveryConvention>(new DateTime2Convention());
```

添加需要在内置约定之前或之后运行的约定时，这将是最常使用的，可在以下位置找到内置约定列表：[ModelConfiguration 命名空间](#)。

还可以删除不希望应用于模型的约定。若要删除约定，请使用 Remove 方法。下面是删除 PluralizingTableNameConvention 的示例。

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
}
```

基于模型的约定

2020/3/11 ·

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

基于模型的约定是基于约定的模型配置的高级方法。大多数情况下，应使用[DbModelBuilder 上的自定义 Code First 约定 API](#)。使用基于模型的约定之前，建议了解约定的 DbModelBuilder API。

基于模型的约定允许创建影响不能通过标准约定进行配置的属性和表的约定。例如，表中的鉴别器列按层次结构模型和独立关联列。

创建约定

创建基于模型的约定的第一步是在管道中选择约定需要应用到模型的时间。有两种类型的模型约定：概念（C-空间）和存储区（S 空间）。C-Space 约定应用于应用程序生成的模型，而 S 空间约定应用于表示数据库的模型的版本，并控制自动生成的列的命名方式。

模型约定是从 `IConceptualModelConvention` 或 `IStoreModelConvention` 扩展的类。这些接口都接受类型为 `MetadataItem` 的泛型类型，该类型用于筛选约定适用的数据类型。

添加约定

添加模型约定的方式与常规约定类相同。在 `OnModelCreating` 方法中，将约定添加到模型的约定列表。

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

public class BlogContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Add<MyModelBasedConvention>();
    }
}
```

还可以使用 `AddBefore<>` 或 `AddAfter<>` 方法，与另一种约定相关添加约定。有关实体框架应用的约定的详细信息，请参阅“备注”部分。

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.AddAfter<IdKeyDiscoveryConvention>(new MyModelBasedConvention());
}
```

示例：鉴别器模型约定

重命名由 EF 生成的列是一个无法使用其他约定 API 进行操作的示例。这种情况下，使用模型约定是唯一的选择。

有关如何使用基于模型的约定配置生成的列的示例，请自定义鉴别器列的命名方式。下面是基于模型的简单约定的一个示例，它将名为“鉴别器”的模型中的每一列都重命名为“EntityType”。这包括开发人员简单命名为“鉴别器”的列。由于“鉴别器”列是生成的列，因此需要在 S 空间中运行。

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

class DiscriminatorRenamingConvention : IStoreModelConvention<EdmProperty>
{
    public void Apply(EdmProperty property, DbModel model)
    {
        if (property.Name == "Discriminator")
        {
            property.Name = "EntityType";
        }
    }
}
```

示例：常规 IA 重命名约定

其他更复杂的基于模型的约定示例是配置独立关联(IAs)的命名方式。这种情况下，模型约定适用，因为 IAs 由 EF 生成，并且不存在于 DbModelBuilder API 可以访问的模型中。

当 EF 生成 IA 时，它将创建一个名为 EntityType_KeyName 的列。例如，对于名为 Customer 且名为 CustomerId 的键列的关联，它会生成一个名为 Customer_CustomerId 的列。下面的约定从为 IA 生成的列名称中去除“_”字符。

```

using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;

// Provides a convention for fixing the independent association (IA) foreign key column names.
public class ForeignKeyNamingConvention : IStoreModelConvention<AssociationType>
{
    public void Apply(AssociationType association, DbModel model)
    {
        // Identify ForeignKey properties (including IAs)
        if (association.IsForeignKey)
        {
            // rename FK columns
            var constraint = association.Constraint;
            if (DoPropertiesHaveDefaultNames(constraint.FromProperties, constraint.ToRole.Name,
                constraint.ToProperties))
            {
                NormalizeForeignKeyProperties(constraint.FromProperties);
            }
            if (DoPropertiesHaveDefaultNames(constraint.ToProperties, constraint.FromRole.Name,
                constraint.FromProperties))
            {
                NormalizeForeignKeyProperties(constraint.ToProperties);
            }
        }
    }

    private bool DoPropertiesHaveDefaultNames(ReadOnlyMetadataCollection<EdmProperty> properties, string
        roleName, ReadOnlyMetadataCollection<EdmProperty> otherEndProperties)
    {
        if (properties.Count != otherEndProperties.Count)
        {
            return false;
        }

        for (int i = 0; i < properties.Count; ++i)
        {
            if (!properties[i].Name.EndsWith("_" + otherEndProperties[i].Name))
            {
                return false;
            }
        }
        return true;
    }

    private void NormalizeForeignKeyProperties(ReadOnlyMetadataCollection<EdmProperty> properties)
    {
        for (int i = 0; i < properties.Count; ++i)
        {
            int underscoreIndex = properties[i].Name.IndexOf('_');
            if (underscoreIndex > 0)
            {
                properties[i].Name = properties[i].Name.Remove(underscoreIndex, 1);
            }
        }
    }
}

```

扩展现有约定

如果需要编写一个约定，此约定与实体框架已应用于你的模型的约定之一类似，你始终可以扩展该约定，以避免必须从头开始重新编写该约定。例如，将现有的 Id 匹配约定替换为自定义约定。提高密钥约定的优点是：只有在未检测到或未显式配置任何键时才会调用重写的方法。实体框架使用的约定列表如下所示：

```
using System.Data.Entity;
using System.Data.Entity.Core.Metadata.Edm;
using System.Data.Entity.Infrastructure;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;

// Convention to detect primary key properties.
// Recognized naming patterns in order of precedence are:
// 1. 'Key'
// 2. [type name]Key
// Primary key detection is case insensitive.
public class CustomKeyDiscoveryConvention : KeyDiscoveryConvention
{
    private const string Id = "Key";

    protected override IEnumerable<EdmProperty> MatchKeyProperty(
        EntityType entityType, IEnumerable<EdmProperty> primitiveProperties)
    {
        Debug.Assert(entityType != null);
        Debug.Assert(primitiveProperties != null);

        var matches = primitiveProperties
            .Where(p => Id.Equals(p.Name, StringComparison.OrdinalIgnoreCase));

        if (!matches.Any())
        {
            matches = primitiveProperties
                .Where(p => (entityType.Name + Id).Equals(p.Name, StringComparison.OrdinalIgnoreCase));
        }

        // If the number of matches is more than one, then multiple properties matched differing only by
        // case--for example, "Key" and "key".
        if (matches.Count() > 1)
        {
            throw new InvalidOperationException("Multiple properties match the key convention");
        }

        return matches;
    }
}
```

接下来，我们需要在现有的密钥约定之前添加新约定。添加 CustomKeyDiscoveryConvention 后，我们可以删除 IdKeyDiscoveryConvention。如果我们未删除现有的 IdKeyDiscoveryConvention，则此约定仍优先于 Id 发现约定，因为它首先运行，但在未找到“密钥”属性的情况下，将运行 “Id” 约定。我们会看到此行为，因为每个约定都将模型视为由上一个约定更新的（而不是单独对其进行操作并将所有组合在一起），因此，如果上一个约定更新了列名以匹配你的自定义约定的兴趣（早于该名称不重要），则它将应用于该列。

```
public class BlogContext : DbContext
{
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.AddBefore<IdKeyDiscoveryConvention>(new CustomKeyDiscoveryConvention());
        modelBuilder.Conventions.Remove<IdKeyDiscoveryConvention>();
    }
}
```

注意

此处的 MSDN 文档提供了实体框架当前应用的约定列表：

<http://msdn.microsoft.com/library/system.data.entity.modelconfiguration.conventions.aspx>。此列表是直接从源代码中提取的。GitHub 上提供了实体框架 6 的源代码，而实体框架所使用的许多约定都是基于自定义模型的约定的良好起点。

熟知 API-关系

2020/3/11 •

NOTE

本页提供有关使用 Fluent API 设置 Code First 模型中的关系的信息。有关 EF 中的关系以及如何使用关系访问和操作数据的一般信息，请参阅[关系 & 导航属性](#)。

使用 Code First 时，可通过定义域 CLR 类定义模型。默认情况下，实体框架使用 Code First 约定将类映射到数据库架构。如果你使用 Code First 命名约定，则在大多数情况下，你可以依赖于 Code First 根据你在类上定义的外键和导航属性来设置表之间的关系。如果在定义类时不遵循约定，或者若要更改约定的工作方式，可以使用 Fluent API 或数据批注来配置类，以便 Code First 可以映射表之间的关系。

介绍

在配置与 Fluent API 的关系时，请从 EntityTypeConfiguration 实例开始，然后使用 HasRequired、HasOptional 或 HasMany 方法来指定此实体参与的关系的类型。HasRequired 和 HasOptional 方法采用表示引用导航属性的 lambda 表达式。HasMany 方法采用表示集合导航属性的 lambda 表达式。然后，可以通过使用 WithRequired、WithOptional 和 WithMany 方法配置反向导航属性。这些方法具有不带参数的重载，可用于通过单向导航指定基数。

然后，可以使用 HasForeignKey 方法配置外键属性。此方法采用一个表示要用作外键的属性的 lambda 表达式。

配置所需的可选关系(一对零或一)

下面的示例将配置一对零或一关系。OfficeAssignment 具有 InstructorID 属性，该属性是主键和外键，因为属性的名称不遵循该约定。HasKey 方法用于配置主键。

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
    .HasRequired(t => t.Instructor)
    .WithOptional(t => t.OfficeAssignment);
```

配置两个端都需要的关系(一对一)

在大多数情况下实体框架可以推断哪个类型是依赖项并且是关系中的主体。但是，如果需要关系的两端，或者两个两侧都是可选的，则实体框架无法识别依赖项和主体。如果关系的两端都是必需的，请在 HasRequired 方法后面使用 WithRequiredPrincipal 或 WithRequiredDependent。如果关系两端都是可选的，请在 HasOptional 方法后面使用 WithOptionalPrincipal 或 WithOptionalDependent。

```
// Configure the primary key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);
```

配置多对多关系

下面的代码配置课程类型和指导员类型之间的多对多关系。在下面的示例中，使用了默认 Code First 约定来创建联接表。因此，Courseinstructor.courseid 表是使用 Course_CourseID 和 Instructor_InstructorID 列创建的。

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
```

如果要使用 Map 方法指定联接表名称和表中列的名称，则需要执行其他配置。下面的代码生成包含 CourseID 和 InstructorID 列的 Courseinstructor.courseid 表。

```
modelBuilder.Entity<Course>()
    .HasMany(t => t.Instructors)
    .WithMany(t => t.Courses)
    .Map(m =>
{
    m.ToTable("CourseInstructor");
    m.MapLeftKey("CourseID");
    m.MapRightKey("InstructorID");
});
```

使用一个导航属性配置关系

单向(也称为单向)关系是指仅在一个关系端上定义导航属性，而不是在两者上定义。按照约定，Code First 始终将单向关系解释为一对多。例如，如果你想要在讲师与 OfficeAssignment 之间进行一对一关系，其中仅有指导员类型的导航属性，则需要使用 Fluent API 来配置此关系。

```
// Configure the primary Key for the OfficeAssignment
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal();
```

启用级联删除

您可以通过使用 WillCascadeOnDelete 方法为关系配置级联删除。如果从属实体上的外键不可为 null，则 Code First 在关系上设置级联删除。如果从属实体上的外键可为 null，则 Code First 不会对关系设置级联删除，并且在删除主体时，外键将设置为 null。

您可以使用以下方法删除这些级联删除约定：

```
modelBuilder<OneToManyCascadeDeleteConvention>()
modelBuilder<ManyToManyCascadeDeleteConvention>()
```

下面的代码将关系配置为 "必需"，然后禁用级联删除。

```
modelBuilder.Entity<Course>()
    .IsRequired(t => t.Department)
    .WithMany(t => t.Courses)
    .HasForeignKey(d => d.DepartmentID)
    .WillCascadeOnDelete(false);
```

配置复合外键

如果部门类型上的主键由 DepartmentID 和 Name 属性组成，则可以为该部门配置主键，并为课程类型配置外键，如下所示：

```
// Composite primary key
modelBuilder.Entity<Department>()
    .HasKey(d => new { d.DepartmentID, d.Name });

// Composite foreign key
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

重命名未在模型中定义的外键

如果选择不在 CLR 类型上定义外键，但要指定它应在数据库中具有的名称，请执行以下操作：

```
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

配置不遵循 Code First 约定的外键名称

如果课程类的外键属性称为 SomeDepartmentID 而不是 DepartmentID，则需要执行以下操作，以指定要 SomeDepartmentID 为外键：

```
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(c => c.SomeDepartmentID);
```

示例中使用的模型

以下 Code First 模型用于此页上的示例。

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
// add a reference to System.ComponentModel.DataAnnotations DLL
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System;

public class SchoolEntities : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
```

```

public DbSet<Instructor> Instructors { get; set; }
public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Configure Code First to ignore PluralizingTableName convention
    // If you keep this convention then the generated tables will have pluralized names.
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
    // Primary key
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public decimal Budget { get; set; }
    public System.DateTime StartDate { get; set; }
    public int? Administrator { get; set; }

    // Navigation property
    public virtual ICollection<Course> Courses { get; private set; }
}

public class Course
{
    public Course()
    {
        this.Instructors = new HashSet<Instructor>();
    }
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
    public virtual ICollection<Instructor> Instructors { get; private set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}

```

```
public class Instructor
{
    public Instructor()
    {
        this.Courses = new List<Course>();
    }

    // Primary key
    public int InstructorID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public System.DateTime HireDate { get; set; }

    // Navigation properties
    public virtual ICollection<Course> Courses { get; private set; }
}

public class OfficeAssignment
{
    // Specifying InstructorID as a primary
    [Key()]
    public Int32 InstructorID { get; set; }

    public string Location { get; set; }

    // When Entity Framework sees Timestamp attribute
    // it configures ConcurrencyCheck and DatabaseGeneratedPattern=Computed.
    [Timestamp]
    public Byte[] Timestamp { get; set; }

    // Navigation property
    public virtual Instructor Instructor { get; set; }
}
```

熟知 API-配置和映射属性和类型

2020/3/11 •

在使用实体框架时 Code First 默认行为是使用一组约定融入为 EF 将 POCO 类映射到表。但有时，您不能或不想遵循这些约定，也不需要将实体映射到约定规定的内容。

可以通过两种主要方式将 EF 配置为使用约定以外的其他内容，即**批注**或 EFs Fluent API。批注仅涵盖 Fluent API 功能的子集，因此存在无法使用批注实现的映射方案。本文旨在演示如何使用 Fluent API 来配置属性。

通过重写派生 `DbContext` 上的 `OnModelCreating` 方法，最常访问的代码优先 Fluent API。下面的示例旨在演示如何使用流畅 api 完成各种任务，并允许你复制代码并对其进行自定义以适合你的模型，如果你希望查看可以按原样使用的模型，则在本文末尾提供。

模型范围内的设置

默认架构 (EF6 向前)

从 EF6 开始，可以对 `DbModelBuilder` 使用 `HasDefaultSchema` 方法，以指定要用于所有表、存储过程等的数据库架构。对于为其显式配置不同架构的任何对象，将重写此默认设置。

```
modelBuilder.HasDefaultSchema("sales");
```

自定义约定 (EF6)

从 EF6 开始，你可以创建自己的约定来补充 Code First 中包含的约定。有关更多详细信息，请参阅[自定义 Code First 约定](#)。

属性映射

属性方法用于为属于实体或复杂类型的每个属性配置特性。属性方法用于获取给定属性的配置对象。配置对象上的选项特定于正在配置的类型。`IsUnicode` 仅可用于字符串属性（例如）。

配置主键

主键的实体框架约定为：

- 类定义的属性的名称为 "ID" 或 "Id"
- 或类名后跟 "ID" 或 "Id"

若要将属性显式设置为主键，可以使用 `HasKey` 方法。在下面的示例中，`HasKey` 方法用于在 `OfficeAssignment` 类型上配置 `InstructorID` 主键。

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

配置组合主键

下面的示例将 `DepartmentID` 和 `Name` 属性配置为部门类型的复合主键。

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

关闭数字主键的标识

下面的示例将 `DepartmentID` 属性设置为 `System.ComponentModel`，以指示数据库将不会生成此值。

```
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

指定属性的最大长度

在下面的示例中，Name 属性的长度不应超过50个字符。如果将值设置为长度超过50个字符，则会收到`DbEntityValidationException`异常。如果 Code First 通过此模型创建数据库，则还会将“名称”列的最大长度设置为50个字符。

```
modelBuilder.Entity<Department>().Property(t => t.Name).HasMaxLength(50);
```

将属性配置为必填

在下面的示例中，Name 属性是必需的。如果未指定名称，则会收到 `DbEntityValidationException` 异常。如果 Code First 通过此模型创建数据库，则用于存储此属性的列通常不能为 null。

NOTE

在某些情况下，即使属性是必需的，也无法使数据库中的列不可为 null。例如，对多个类型使用 TPH 继承战略数据时，会将其存储在一个表中。如果派生的类型包含所需的属性，则列不能为 null，因为并不是层次结构中的所有类型都具有此属性。

```
modelBuilder.Entity<Department>().Property(t => t.Name).IsRequired();
```

对一个或多个属性配置索引

NOTE

Ef 6.1 ■往上-索引属性是在实体框架6.1 中引入的。如果你使用的是早期版本，则本部分中的信息不适用。

流畅的 API 不支持创建索引，但你可以通过流畅的 API 使用对`IndexAttribute`的支持。索引属性的处理方式是在模型中包含模型批注，然后在管道中的后续部分将其转换为数据库中的索引。可以使用熟知的 API 手动添加这些相同的注释。

执行此操作的最简单方法是创建一个`IndexAttribute`实例，其中包含新索引的所有设置。然后，你可以创建一个`IndexAnnotation`实例，该实例是一个 EF 特定类型，它将`IndexAttribute`设置转换为可存储在 EF 模型上的模型注释。然后，可以将这些方法传递到熟知 API 上的`HasColumnAnnotation`方法，并指定该批注的名称索引。

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute()));
```

有关`IndexAttribute`中可用设置的完整列表，请参阅[Code First 数据批注](#)的索引部分。这包括自定义索引名称、创建唯一索引以及创建多列索引。

可以通过将`IndexAttribute`数组传递到`IndexAnnotation`的构造函数，在单个属性上指定多个索引批注。

```
modelBuilder
    .Entity<Department>()
    .Property(t => t.Name)
    .HasColumnAnnotation(
        "Index",
        new IndexAnnotation(new[]
        {
            new IndexAttribute("Index1"),
            new IndexAttribute("Index2") { IsUnique = true }
        }));
    
```

指定不将 CLR 属性映射到数据库中的列

下面的示例演示如何指定 CLR 类型的属性未映射到数据库中的列。

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

将 CLR 属性映射到数据库中的特定列

下面的示例将 Name CLR 属性映射到 DepartmentName 数据库列。

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

重命名未在模型中定义的外键

如果选择不在 CLR 类型上定义外键，但要指定它应在数据库中具有的名称，请执行以下操作：

```
modelBuilder.Entity<Course>()
    .IsRequired(c => c.Department)
    .WithMany(t => t.Courses)
    .Map(m => m.MapKey("ChangedDepartmentID"));
```

配置字符串属性是否支持 Unicode 内容

默认情况下，字符串为 Unicode (SQL Server 中为 nvarchar)。可以使用 IsUnicode 方法来指定字符串应为 varchar 类型。

```
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .IsUnicode(false);
```

配置数据库列的数据类型

[HasColumnType](#)方法允许映射到相同基本类型的不同表示形式。使用此方法不能在运行时执行数据的任何转换。请注意，IsUnicode 是将列设置为 varchar 的首选方式，因为它是数据库不可知的。

```
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

配置复杂类型的属性

可以通过两种方法来配置复杂类型的标量属性。

可以在 [ComplexTypeConfiguration](#) 上调用属性。

```
modelBuilder.ComplexType<Details>()
    .Property(t => t.Location)
    .HasMaxLength(20);
```

你还可以使用点表示法访问复杂类型的属性。

```
modelBuilder.Entity<OnsiteCourse>()
    .Property(t => t.Details.Location)
    .HasMaxLength(20);
```

配置要用作开放式并发标记的属性

若要指定实体中的属性表示并发标记，可以使用 `ConcurrencyCheck` 特性或 `IsConcurrencyToken` 方法。

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

还可以使用 `IsRowVersion` 方法将属性配置为数据库中的行版本。将属性设置为行版本会自动将它配置为开放式并发标记。

```
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsRowVersion();
```

类型映射

指定类为复杂类型

按照约定，未指定主键的类型将被视为复杂类型。在某些情况下，Code First 不会检测到复杂类型（例如，如果你有一个名为 `ID` 的属性，但你并不意味着它是主键）。在这种情况下，可以使用 Fluent API 显式指定类型为复杂类型。

```
modelBuilder.ComplexType<Details>();
```

指定不将 CLR 实体类型映射到数据库中的表

下面的示例演示如何将 CLR 类型从映射到数据库中的表。

```
modelBuilder.Ignore<OnlineCourse>();
```

将实体类型映射到数据库中的特定表

部门的所有属性都将映射到名为 `t_Department` 的表中的列。

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

你还可以指定架构名称，如下所示：

```
modelBuilder.Entity<Department>()
    .ToTable("t_Department", "school");
```

映射每个层次结构一个表 (TPH) 继承

在 TPH 映射方案中，继承层次结构中的所有类型都映射到单个表。鉴别器列用于标识每行的类型。在 Code First 创建模型时，TPH 是参与继承层次结构的类型的默认策略。默认情况下，鉴别器列将添加到名称为 "鉴别器" 的表中，层次结构中的每个类型的 CLR 类型名称将用于鉴别器值。您可以使用 Fluent API 修改默认行为。

```
modelBuilder.Entity<Course>()
    .Map<Course>(m => m.Requires("Type").HasValue("Course"))
    .Map<OnsiteCourse>(m => m.Requires("Type").HasValue("OnsiteCourse"));
```

映射每种类型一个表(TPT)继承

在 TPT 映射方案中，所有类型都映射到单个表。仅属于某个基类型或派生类型的属性存储在映射到该类型的一个表中。映射到派生类型的表还存储将派生表与基表联接的外键。

```
modelBuilder.Entity<Course>().ToTable("Course");
modelBuilder.Entity<OnsiteCourse>().ToTable("OnsiteCourse");
```

映射每个具体的表类(TPC)继承

在 TPC 映射方案中，层次结构中的所有非抽象类型都将映射到单个表。映射到派生类的表与映射到数据库中的基类的表没有关系。类的所有属性（包括继承的属性）都映射到对应表的列。

调用 MapInheritedProperties 方法来配置每个派生类型。对于派生类，MapInheritedProperties 将从基类继承的所有属性重新映射到表中的新列。

NOTE

请注意，由于参与了 TPC 继承层次结构的表不共享主键，因此，如果数据库生成的值具有相同的标识种子，则在映射到子类的表中插入时，将出现重复的实体键。若要解决此问题，可以为每个表指定不同的初始种子值，或者在 primary key 属性上关闭“标识”。当使用 Code First 时，标识是整数键属性的默认值。

```
modelBuilder.Entity<Course>()
    .Property(c => c.CourseID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

modelBuilder.Entity<OnsiteCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnsiteCourse");
});

modelBuilder.Entity<OnlineCourse>().Map(m =>
{
    m.MapInheritedProperties();
    m.ToTable("OnlineCourse");
});
```

将实体类型的属性映射到数据库中的多个表(实体拆分)

实体拆分允许将某个实体类型的属性分布到多个表中。在下面的示例中，部门实体拆分为两个表：部门和 DepartmentDetails。实体拆分对 Map 方法使用多个调用，以将属性的子集映射到特定的表。

```
modelBuilder.Entity<Department>()
    .Map(m =>
{
    m.Properties(t => new { t.DepartmentID, t.Name });
    m.ToTable("Department");
})
.Map(m =>
{
    m.Properties(t => new { t.DepartmentID, t.Administrator, t.StartDate, t.Budget });
    m.ToTable("DepartmentDetails");
});
```

将多个实体类型映射到数据库中的一个表(表拆分)

下面的示例将共享主键的两个实体类型映射到一个表。

```
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

将实体类型映射到插入/更新/删除存储过程(EF6)

从 EF6 开始，可以将实体映射为使用存储过程来执行插入更新和删除操作。有关更多详细信息，请参阅[Code First 插入/更新/删除存储过程](#)。

示例中使用的模型

以下 Code First 模型用于此页上的示例。

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
// add a reference to System.ComponentModel.DataAnnotations DLL
using System.ComponentModel.DataAnnotations;
using System.Collections.Generic;
using System;

public class SchoolEntities : DbContext
{
    public DbSet<Course> Courses { get; set; }
    public DbSet<Department> Departments { get; set; }
    public DbSet<Instructor> Instructors { get; set; }
    public DbSet<OfficeAssignment> OfficeAssignments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Configure Code First to ignore PluralizingTableName convention
        // If you keep this convention then the generated tables will have pluralized names.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}

public class Department
{
    public Department()
    {
        this.Courses = new HashSet<Course>();
    }
```

```

        // Primary key
        public int DepartmentID { get; set; }
        public string Name { get; set; }
        public decimal Budget { get; set; }
        public System.DateTime StartDate { get; set; }
        public int? Administrator { get; set; }

        // Navigation property
        public virtual ICollection<Course> Courses { get; private set; }
    }

public class Course
{
    public Course()
    {
        this.Instructors = new HashSet<Instructor>();
    }
    // Primary key
    public int CourseID { get; set; }

    public string Title { get; set; }
    public int Credits { get; set; }

    // Foreign key
    public int DepartmentID { get; set; }

    // Navigation properties
    public virtual Department Department { get; set; }
    public virtual ICollection<Instructor> Instructors { get; private set; }
}

public partial class OnlineCourse : Course
{
    public string URL { get; set; }
}

public partial class OnsiteCourse : Course
{
    public OnsiteCourse()
    {
        Details = new Details();
    }

    public Details Details { get; set; }
}

public class Details
{
    public System.DateTime Time { get; set; }
    public string Location { get; set; }
    public string Days { get; set; }
}

public class Instructor
{
    public Instructor()
    {
        this.Courses = new List<Course>();
    }

    // Primary key
    public int InstructorID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public System.DateTime HireDate { get; set; }

    // Navigation properties
    public virtual ICollection<Course> Courses { get; private set; }
}
```

```
}

public class OfficeAssignment
{
    // Specifying InstructorID as a primary
    [Key()]
    public Int32 InstructorID { get; set; }

    public string Location { get; set; }

    // When Entity Framework sees Timestamp attribute
    // it configures ConcurrencyCheck and DatabaseGeneratedPattern=Computed.
    [Timestamp]
    public Byte[] Timestamp { get; set; }

    // Navigation property
    public virtual Instructor Instructor { get; set; }
}
```

通过 VB.NET 的流畅 API

2020/3/11 •

Code First 允许使用 C# 或 VB.NET 类定义模型。还可以选择使用类和属性上的属性或使用 Fluent API 来执行其他配置。本演练演示如何使用 VB.NET 执行 Fluent API 配置。

本页假设您基本了解 Code First。有关 Code First 的详细信息，请参阅以下演练：

- [对新数据库使用 Code First](#)
- [Code First 到现有数据库](#)

先决条件

需要至少安装 Visual Studio 2010 或 Visual Studio 2012 才能完成此演练。

如果你使用的是 Visual Studio 2010，你还需要安装[NuGet](#)

创建应用程序

为了简单起见，我们将构建一个使用 Code First 执行数据访问的基本控制台应用程序。

- 打开 Visual Studio
- 文件->> 项目。
- 从左侧菜单和控制台应用程序选择Windows
- 输入CodeFirstVBSSample作为名称
- 选择“确定”

定义模型

在此步骤中，将定义表示概念模型的 VB.NET POCO 实体类型。类不需要从任何基类派生或实现任何接口。

- 向项目中添加一个新类，为类名输入SchoolModel
- 将新类的内容替换为以下代码

```
Public Class Department
    Public Sub New()
        Me.Courses = New List(Of Course)()
    End Sub

    ' Primary key
    Public Property DepartmentID() As Integer
    Public Property Name() As String
    Public Property Budget() As Decimal
    Public Property StartDate() As Date
    Public Property Administrator() As Integer?
    Public Overridable Property Courses() As ICollection(Of Course)
End Class

Public Class Course
    Public Sub New()
        Me.Instructors = New HashSet(Of Instructor)()
    End Sub

    ' Primary key
    Public Property CourseID() As Integer
    Public Property Title() As String
```

```

    Public Property Credits() As Integer

        ' Foreign key that does not follow the Code First convention.
        ' The fluent API will be used to configure DepartmentID_FK to be the foreign key for this entity.
        Public Property DepartmentID_FK() As Integer

        ' Navigation properties
        Public Overridable Property Department() As Department
        Public Overridable Property Instructors() As ICollection(Of Instructor)
End Class

Public Class OnlineCourse
    Inherits Course

    Public Property URL() As String
End Class

Partial Public Class OnsiteCourse
    Inherits Course

    Public Sub New()
        Details = New OnsiteCourseDetails()
    End Sub

    Public Property Details() As OnsiteCourseDetails
End Class

' Complex type
Public Class OnsiteCourseDetails
    Public Property Time() As Date
    Public Property Location() As String
    Public Property Days() As String
End Class

Public Class Person
    ' Primary key
    Public Property PersonID() As Integer
    Public Property LastName() As String
    Public Property FirstName() As String
End Class

Public Class Instructor
    Inherits Person

    Public Sub New()
        Me.Courses = New List(Of Course)()
    End Sub

    Public Property HireDate() As Date

    ' Navigation properties
    Private privateCourses As ICollection(Of Course)
    Public Overridable Property Courses() As ICollection(Of Course)
    Public Overridable Property OfficeAssignment() As OfficeAssignment
End Class

Public Class OfficeAssignment
    ' Primary key that does not follow the Code First convention.
    ' The HasKey method is used later to configure the primary key for the entity.
    Public Property InstructorID() As Integer

    Public Property Location() As String
    Public Property Timestamp() As Byte()

    ' Navigation property
    Public Overridable Property Instructor() As Instructor
End Class

```

定义派生上下文

我们即将开始使用实体框架中的类型，因此我们需要添加 EntityFramework NuGet 包。

- ** 项目->管理 NuGet 程序包 ...

NOTE

如果你没有 "NuGet ..." 选项你应安装[最新版本的 NuGet](#)

- 选择 "联机" 选项卡
- 选择 EntityFramework 包
- 单击"安装"

现在可以定义一个派生上下文，它表示与数据库的会话，从而使我们能够查询和保存数据。我们定义了从 DbContext 派生的上下文，并为模型中的每个类公开了类型化的 DbSet< TEntity >。

- 向项目中添加一个新类，为类名输入 SchoolContext
- 将新类的内容替换为以下代码

```
Imports System.ComponentModel.DataAnnotations
Imports System.ComponentModel.DataAnnotations.Schema
Imports System.Data.Entity
Imports System.Data.Entity.Infrastructure
Imports System.Data.Entity.ModelConfiguration.Conventions

Public Class SchoolContext
    Inherits DbContext

    Public Property OfficeAssignments() As DbSet(Of OfficeAssignment)
    Public Property Instructors() As DbSet(Of Instructor)
    Public Property Courses() As DbSet(Of Course)
    Public Property Departments() As DbSet(Of Department)

    Protected Overrides Sub OnModelCreating(ByVal modelBuilder As DbModelBuilder)
        End Sub
    End Class
```

用熟知 API 配置

本部分演示如何使用熟知的 API 将类型配置为表映射、属性到列的映射，以及模型中\类型的表之间的关系。Fluent API 通过 DbModelBuilder 类型公开，最常用的方法是重写 DbContext 上的 OnModelCreating 方法。

- 复制以下代码并将其添加到 SchoolContext 类中定义的 OnModelCreating 方法中。注释说明了每个映射的作用

```
' Configure Code First to ignore PluralizingTableName convention
' If you keep this convention then the generated tables
' will have pluralized names.
modelBuilder.Conventions.Remove(Of PluralizingTableNameConvention)()

' Specifying that a Class is a Complex Type

' The model defined in this topic defines a type OnsiteCourseDetails.
' By convention, a type that has no primary key specified
' is treated as a complex type.
' There are some scenarios where Code First will not
' detect a complex type (for example, if you do have a property
```

```

' called ID, but you do not mean for it to be a primary key).
' In such cases, you would use the fluent API to
' explicitly specify that a type is a complex type.
modelBuilder.ComplexType(Of OnsiteCourseDetails)()

' Mapping a CLR Entity Type to a Specific Table in the Database.

' All properties of OfficeAssignment will be mapped
' to columns in a table called t_OfficeAssignment.
modelBuilder.Entity(Of OfficeAssignment)().ToTable("t_OfficeAssignment")

' Mapping the Table-Per-Hierarchy (TPH) Inheritance

' In the TPH mapping scenario, all types in an inheritance hierarchy
' are mapped to a single table.
' A discriminator column is used to identify the type of each row.
' When creating your model with Code First,
' TPH is the default strategy for the types that
' participate in the inheritance hierarchy.
' By default, the discriminator column is added
' to the table with the name "Discriminator"
' and the CLR type name of each type in the hierarchy
' is used for the discriminator values.
' You can modify the default behavior by using the fluent API.
modelBuilder.Entity(Of Person)().
    Map(Of Person)(Function(t) t.Requires("Type").
        HasValue("Person")).
    Map(Of Instructor)(Function(t) t.Requires("Type").
        HasValue("Instructor"))

' Mapping the Table-Per-Type (TPT) Inheritance

' In the TPT mapping scenario, all types are mapped to individual tables.
' Properties that belong solely to a base type or derived type are stored
' in a table that maps to that type. Tables that map to derived types
' also store a foreign key that joins the derived table with the base table.
modelBuilder.Entity(Of Course)().ToTable("Course")
modelBuilder.Entity(Of OnsiteCourse)().ToTable("OnsiteCourse")
modelBuilder.Entity(Of OnlineCourse)().ToTable("OnlineCourse")

' Configuring a Primary Key

' If your class defines a property whose name is "ID" or "Id",
' or a class name followed by "ID" or "Id",
' the Entity Framework treats this property as a primary key by convention.
' If your property name does not follow this pattern, use the HasKey method
' to configure the primary key for the entity.
modelBuilder.Entity(Of OfficeAssignment)().
    HasKey(Function(t) t.InstructorID)

' Specifying the Maximum Length on a Property

' In the following example, the Name property
' should be no longer than 50 characters.
' If you make the value longer than 50 characters,
' you will get a DbEntityValidationException exception.
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    HasMaxLength(60)

' Configuring the Property to be Required

' In the following example, the Name property is required.
' If you do not specify the Name,

```

```

' you will get a DbEntityValidationException exception.
' The database column used to store this property will be non-nullable.
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    IsRequired()

' Switching off Identity for Numeric Primary Keys

' The following example sets the DepartmentID property to
' System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None to indicate that
' the value will not be generated by the database.
modelBuilder.Entity(Of Course)().Property(Function(t) t.CourseID).
    HasDatabaseGeneratedOption(DatabaseGeneratedOption.None)

'Specifying NOT to Map a CLR Property to a Column in the Database
modelBuilder.Entity(Of Department)().
    Ignore(Function(t) t.Administrator)

'Mapping a CLR Property to a Specific Column in the Database
modelBuilder.Entity(Of Department)().Property(Function(t) t.Budget).
    HasColumnName("DepartmentBudget")

'Configuring the Data Type of a Database Column
modelBuilder.Entity(Of Department)().Property(Function(t) t.Name).
    HasColumnType("varchar")

'Configuring Properties on a Complex Type
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Days).
    HasColumnName("Days")
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Location).
    HasColumnName("Location")
modelBuilder.Entity(Of OnsiteCourse)().Property(Function(t) t.Details.Time).
    HasColumnName("Time")

' Map one-to-zero or one relationship

' The OfficeAssignment has the InstructorID
' property that is a primary key and a foreign key.
modelBuilder.Entity(Of OfficeAssignment)().
    HasRequired(Function(t) t.Instructor).
    WithOptional(Function(t) t.OfficeAssignment)

' Configuring a Many-to-Many Relationship

' The following code configures a many-to-many relationship
' between the Course and Instructor types.
' In the following example, the default Code First conventions
' are used to create a join table.
' As a result the CourseInstructor table is created with
' Course_CourseID and Instructor_InstructorID columns.
modelBuilder.Entity(Of Course)().
    HasMany(Function(t) t.Instructors).
    WithMany(Function(t) t.Courses)

' Configuring a Many-to-Many Relationship and specifying the names
' of the columns in the join table

' If you want to specify the join table name
' and the names of the columns in the table
' you need to do additional configuration by using the Map method.
' The following code generates the CourseInstructor
' table with CourseID and InstructorID columns.
modelBuilder.Entity(Of Course)().
    HasMany(Function(t) t.Instructors).
    WithMany(Function(t) t.Courses).
    Map(Sub(m)

```

```

    m.ToTable("CourseInstructor")
    m.MapLeftKey("CourseID")
    m.MapRightKey("InstructorID")
End Sub)

' Configuring a foreign key name that does not follow the Code First convention

' The foreign key property on the Course class is called DepartmentID_FK
' since that does not follow Code First conventions you need to explicitly specify
' that you want DepartmentID_FK to be the foreign key.
modelBuilder.Entity(Of Course)().
    HasRequired(Function(t) t.Department).
    WithMany(Function(t) t.Courses).
    HasForeignKey(Function(t) t.DepartmentID_FK)

' Enabling Cascade Delete

' By default, if a foreign key on the dependent entity is not nullable,
' then Code First sets cascade delete on the relationship.
' If a foreign key on the dependent entity is nullable,
' Code First does not set cascade delete on the relationship,
' and when the principal is deleted the foreign key will be set to null.
' The following code configures cascade delete on the relationship.

' You can also remove the cascade delete conventions by using:
' modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()
' and modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>().
modelBuilder.Entity(Of Course)().
    HasRequired(Function(t) t.Department).
    WithMany(Function(t) t.Courses).
    HasForeignKey(Function(d) d.DepartmentID_FK).
    WillCascadeOnDelete(False)

```

使用模型

让我们使用SchoolContext执行一些数据访问，以查看操作中的模型。

- 打开定义了 Main 函数的 Module1 文件
- 复制并粘贴以下 Module1 定义

```
Imports System.Data.Entity

Module Module1

Sub Main()

    Using context As New SchoolContext()

        ' Create and save a new Department.
        Console.Write("Enter a name for a new Department: ")
        Dim name = Console.ReadLine()

        Dim department = New Department With { .Name = name, .StartDate = DateTime.Now }
        context.Departments.Add(department)
        context.SaveChanges()

        ' Display all Departments from the database ordered by name
        Dim departments =
            From d In context.Departments
            Order By d.Name
            Select d

        Console.WriteLine("All Departments in the database:")
        For Each department In departments
            Console.WriteLine(department.Name)
        Next

    End Using

    Console.WriteLine("Press any key to exit...")
    Console.ReadKey()

End Sub

End Module
```

你现在可以运行该应用程序并对其进行测试。

```
Enter a name for a new Department: Computing
All Departments in the database:
Computing
Press any key to exit...
```

Code First 插入、更新和删除存储过程

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

默认情况下，Code First 会将所有实体配置为使用直接表访问来执行插入、更新和删除命令。从 EF6 开始，你可以配置 Code First 模型，以便为模型中的部分或所有实体使用存储过程。

基本实体映射

可以使用熟知的 API 选择使用存储过程执行插入、更新和删除操作。

```
modelBuilder  
    .Entity<Blog>()  
    .MapToStoredProcedures();
```

这样做将导致 Code First 使用一些约定来生成数据库中的存储过程的预期形状。

- 三个名为 <type_name>_Insert<type_name>_Update<type_name>_Delete Blog_Insert Blog_Update Blog_Delete 的存储过程。
- 参数名称对应于属性名称。

NOTE

如果使用 HasColumnName() 或 Column 特性重命名给定属性的列，则此名称用于参数而不是属性名称。

- Insert 存储过程将为每个属性提供一个参数，但标记为“生成的存储”的属性除外（标识或计算）。存储过程应该返回一个结果集，其中包含每个商店生成的属性的列。
- Update 存储过程的每个属性都有一个参数，但使用存储生成模式“计算”的标记除外。某些并发令牌需要原始值的参数，有关详细信息，请参阅下面的并发标记部分。存储过程应该返回一个结果集，其中包含每个计算属性的列。
- Delete 存储过程应具有实体的键值的参数（如果该实体具有组合键，则为多个参数）。此外，删除过程还应包含目标表上的任何独立关联外键的参数（实体中没有声明的相应外键属性的关系）。某些并发令牌需要原始值的参数，有关详细信息，请参阅下面的并发标记部分。

使用以下类作为示例：

```
public class Blog  
{  
    public int BlogId { get; set; }  
    public string Name { get; set; }  
    public string Url { get; set; }  
}
```

默认存储过程是：

```
CREATE PROCEDURE [dbo].[Blog_Insert]
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
BEGIN
    INSERT INTO [dbo].[Blogs] ([Name], [Url])
    VALUES (@Name, @Url)

    SELECT SCOPE_IDENTITY() AS BlogId
END
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
UPDATE [dbo].[Blogs]
SET [Name] = @Name, [Url] = @Url
WHERE BlogId = @BlogId;
CREATE PROCEDURE [dbo].[Blog_Delete]
    @BlogId int
AS
DELETE FROM [dbo].[Blogs]
WHERE BlogId = @BlogId
```

重写默认值

您可以覆盖默认情况下配置的部分或全部内容。

您可以更改一个或多个存储过程的名称。此示例仅重命名更新存储过程。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog")));
    
```

此示例将重命名所有三个存储过程。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog"))
        .Delete(d => d.HasName("delete_blog"))
        .Insert(i => i.HasName("insert_blog")));
    
```

在这些示例中，调用链接在一起，但你也可以使用 lambda 块语法。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
    {
        s.Update(u => u.HasName("modify_blog"));
        s.Delete(d => d.HasName("delete_blog"));
        s.Insert(i => i.HasName("insert_blog"));
    });
    
```

此示例将重命名更新存储过程中的 BlogId 属性的参数。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.Parameter(b => b.BlogId, "blog_id")));

```

这些调用都是可链且可组合的。下面的示例将重命名所有三个存储过程及其参数。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.HasName("modify_blog")
            .Parameter(b => b.BlogId, "blog_id")
            .Parameter(b => b.Name, "blog_name")
            .Parameter(b => b.Url, "blog_url"))
        .Delete(d => d.HasName("delete_blog")
            .Parameter(b => b.BlogId, "blog_id"))
        .Insert(i => i.HasName("insert_blog")
            .Parameter(b => b.Name, "blog_name")
            .Parameter(b => b.Url, "blog_url")));

```

您还可以更改包含数据库生成值的结果集中的列的名称。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Result(b => b.BlogId, "generated_blog_identity")));

```

```
CREATE PROCEDURE [dbo].[Blog_Insert]
    @Name nvarchar(max),
    @Url nvarchar(max)
AS
BEGIN
    INSERT INTO [dbo].[Blogs] ([Name], [Url])
    VALUES (@Name, @Url)

    SELECT SCOPE_IDENTITY() AS generated_blog_id
END

```

类中没有外键的关系(独立关联)

在类定义中包括外键属性时，可以使用与任何其他属性相同的方式重命名相应的参数。如果存在类中没有外键属性的关系，则默认参数名称为 `<navigation_property_name>_<primary_key_name>`。

例如，以下类定义将导致在要插入和更新发布的存储过程中出现 `Blog_BlogId` 参数。

```

public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

```

重写默认值

通过向参数方法提供 primary key 属性的路径，可以更改未包含在类中的外键的参数。

```

modelBuilder
    .Entity<Post>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Parameter(p => p.Blog.BlogId, "blog_id")));

```

如果依赖实体上没有导航属性(即不是 Post。博客属性)然后，可以使用 Association 方法标识关系的另一端，然后配置与每个键属性对应的参数。

```

modelBuilder
    .Entity<Post>()
    .MapToStoredProcedures(s =>
        s.Insert(i => i.Navigation<Blog>(
            b => b.Posts,
            c => c.Parameter(b => b.BlogId, "blog_id")));

```

并发标记

更新和删除存储过程可能还需要处理并发：

- 如果实体包含并发标记，则存储过程可以有选择性地包含一个 output 参数，该参数返回更新/删除的行数(受影响的行数)。必须使用 RowsAffectedParameter 方法配置此类参数。
默认情况下，EF 使用 ExecuteNonQuery 的返回值来确定受影响的行数。如果在过程中执行的任何逻辑会导致在执行结束时 ExecuteNonQuery 的返回值不正确(从 EF 的角度)，则指定 rows 受影响的输出参数将很有用。
- 对于每个并发标记，都将有一个名为 <property_name>_Original 的参数(例如 Timestamp_Original)。这会传递到此属性的原始值–从数据库查询时的值。
 - 数据库计算的并发性标记(如时间戳)将只有原始值参数。
 - 设置为并发令牌的非计算属性也会在更新过程中为新值提供一个参数。这对新值使用已讨论的命名约定。此类标记的一个示例将使用博客的 URL 作为并发标记，新值是必需的，因为你的代码可以将此值更新为新值(与仅由数据库更新的时间戳标记不同)。

这是一个示例类，并用时间戳并发标记更新存储过程。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    [Timestamp]
    public byte[] Timestamp { get; set; }
}
```

```
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max),
    @Timestamp_Original rowversion
AS
    UPDATE [dbo].[Blogs]
    SET [Name] = @Name, [Url] = @Url
    WHERE BlogId = @BlogId AND [Timestamp] = @Timestamp_Original
```

下面是一个示例类，并使用非计算并发标记更新存储过程。

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    [ConcurrencyCheck]
    public string Url { get; set; }
}
```

```
CREATE PROCEDURE [dbo].[Blog_Update]
    @BlogId int,
    @Name nvarchar(max),
    @Url nvarchar(max),
    @Url_Original nvarchar(max),
AS
    UPDATE [dbo].[Blogs]
    SET [Name] = @Name, [Url] = @Url
    WHERE BlogId = @BlogId AND [Url] = @Url_Original
```

重写默认值

您可以选择引入受影响的行参数。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.RowsAffectedParameter("rows_affected")));
```

对于数据库计算的并发标记(仅传递原始值)，只需使用标准参数重命名机制来重命名原始值的参数。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s =>
        s.Update(u => u.Parameter(b => b.Timestamp, "blog_timestamp")));
```

对于非计算并发标记(同时传递原始值和新值)，可以使用参数的重载，以便为每个参数提供名称。

```
modelBuilder
    .Entity<Blog>()
    .MapToStoredProcedures(s => s.Update(u => u.Parameter(b => b.Url, "blog_url", "blog_original_url")));
```

多对多关系

我们将使用以下类作为此部分中的示例。

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public List<Tag> Tags { get; set; }
}

public class Tag
{
    public int TagId { get; set; }
    public string TagName { get; set; }

    public List<Post> Posts { get; set; }
}
```

多对多关系可以映射到具有以下语法的存储过程。

```
modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(t => t.Posts)
    .MapToStoredProcedures();
```

如果未提供其他配置，则默认情况下使用以下存储过程形状。

- 两个名为 <type_one><type_two>_Insert<type_one> <type_two>_Delete PostTag_Insert PostTag_Delete 和的存储过程。
- 参数将是每个类型的键值。要<的每个参数的名称type_name>_<property_name>（例如 Post_PostId 和 Tag_TagId）。

下面是 insert 和 update 存储过程的示例。

```
CREATE PROCEDURE [dbo].[PostTag_Insert]
    @Post_PostId int,
    @Tag_TagId int
AS
    INSERT INTO [dbo].[Post_Tags] (Post_PostId, Tag_TagId)
    VALUES (@Post_PostId, @Tag_TagId)
CREATE PROCEDURE [dbo].[PostTag_Delete]
    @Post_PostId int,
    @Tag_TagId int
AS
    DELETE FROM [dbo].[Post_Tags]
    WHERE Post_PostId = @Post_PostId AND Tag_TagId = @Tag_TagId
```

重写默认值

可以采用类似的方式配置过程和参数名称以与实体存储过程类似。

```
modelBuilder
    .Entity<Post>()
    .HasMany(p => p.Tags)
    .WithMany(t => t.Posts)
    .MapToStoredProcedures(s =>
        s.Insert(i => i.HasName("add_post_tag")
            .LeftKeyParameter(p => p.PostId, "post_id")
            .RightKeyParameter(t => t.TagId, "tag_id")))
        .Delete(d => d.HasName("remove_post_tag")
            .LeftKeyParameter(p => p.PostId, "post_id")
            .RightKeyParameter(t => t.TagId, "tag_id")));
    
```

Code First 迁移

2020/3/11 •

如果使用的是 Code First 工作流，推荐使用 Code First 迁移改进应用程序的数据库架构。迁移提供一组允许以下操作的工具：

1. 创建可用于 EF 模型的初始数据库
2. 生成迁移以跟踪对 EF 模型所做的更改
3. 使数据库随时掌握这些更改

下方演练将概述实体框架中的 Code First 迁移。可以完成整个演练或跳到感兴趣的主题。包含以下主题：

生成初始模型和数据库

开始使用迁移之前，需要会用到项目和 Code First 模型。对于此演练，我们将使用规范的“博客”和“帖子”模型。

- 创建新的 MigrationsDemo 控制台应用程序
- 将最新版本的 EntityFramework NuGet 包添加到项目中
 - “工具”->“库包管理器”->“包管理器控制台”
 - 运行 Install-Package EntityFramework 命令
- 添加 Model.cs 文件，其代码如下所示。此代码定义了构成域模型的单个“博客”类和 EF Code First 上下文 BlogContext 类

```
using System.Data.Entity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.Infrastructure;

namespace MigrationsDemo
{
    public class BlogContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

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

- 现在我们拥有一个模型，可用它执行数据访问操作。更新 Program.cs 文件，其代码如下所示。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

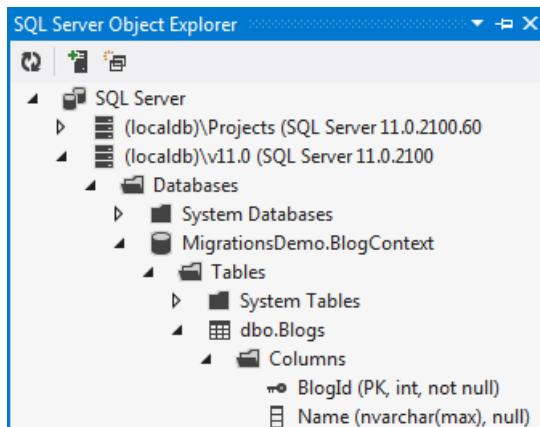
namespace MigrationsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog" });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

- 运行应用程序，随即会创建 MigrationsCodeDemo.BlogContext 数据库。



启用迁移

现可对模型进行更多更改。

- 将 Url 属性引入“博客”类。

```
public string Url { get; set; }
```

如果要再次运行应用程序，则会收到一个 InvalidOperationException，指出“创建数据库后，支持‘BlogContext’上下文的模型已发生变化。请考虑使用 Code First 迁移更新数据库 (<http://go.microsoft.com/fwlink/?LinkId=238269>)。

如异常情况所述，可开始使用 Code First 迁移。第一步是启用上下文迁移。

- 在包管理器控制台中运行 Enable-Migrations 命令

此命令已将“迁移”文件夹添加到项目中。此新文件夹包含两个文件：

- 配置类。此类允许配置迁移对上下文的行为方式。对于此演练，将只使用默认配置。由于项目中只有一个 Code First 上下文，因此 Enable-Migrations 已自动填充此配置适用的上下文类型。
- InitialCreate 迁移。之所以生成此迁移，是因为在启用迁移之前，我们已使用 Code First 创建了数据库。已构建的迁移中的代码表示已在数据库中创建的对象。在本演练中，即为具有 BlogId 和“名称”列的 Blog 表。文件名包含时间戳，这样有助于排序。如果尚未创建数据库，则不会将此 InitialCreate 迁移添加到项目中。相反，第一次调用 Add-Migration 时，会将创建这些表的代码构建到新的迁移中。

针对同一数据库的多个模型

使用 EF6 之前的版本时，只能使用一个 Code First 模型生成/管理数据库的架构。这是因为每个数据库的单个 __MigrationsHistory 表无法识别哪些项属于哪个模型。

从 EF6 开始，配置类中包括 ContextKey 属性。该属性充当每个 Code First 模型的唯一标识符。__MigrationsHistory 表中相应的列允许来自多个模型的项共享表。默认情况下，此属性设置为上下文的完全限定名称。

生成和运行迁移

Code First 迁移具有两个需要用户了解的主要命令。

- Add-Migration 将基于自上次迁移创建以来对模型所做的更改来构建下一次迁移
- Update-Database 将对数据库应用任意挂起的迁移

我们需要构建迁移来处理添加的新 Url 属性。Add-Migration 命令可为这些迁移命名，仅需调用 AddBlogUrl。

- 在包管理器控制台中运行 Add-Migration AddBlogUrl 命令
- 现“迁移”文件夹中具有新的 AddBlogUrl 迁移。迁移文件名以时间戳作为前缀，这样有助于排序

```
namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddBlogUrl : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Blogs", "Url", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Blogs", "Url");
        }
    }
}
```

现可编辑或添加到此迁移，但所有内容看起来都很合适。使用 Update-Database 将此迁移应用到数据库。

- 在包管理器控制台运行 Update-Database 命令
- Code First 迁移将比较“迁移”文件夹中的迁移和已应用于数据库的迁移。迁移会发现需应用 AddBlogUrl 迁移，并运行它。

MigrationsDemo.BlogContext 数据库现已更新，其中包含“博客”表中的 Url 列。

自定义迁移

到目前为止，我们已在未进行任何更改的情况下生成并运行了迁移。现在我们来看看如何编辑默认生成的代码。

- 现在可对模型进行更多更改，我们将新的 Rating 属性添加到“博客”类

```
public int Rating { get; set; }
```

- 同时添加一个新的“帖子”类

```
public class Post
{
    public int PostId { get; set; }
    [MaxLength(200)]
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

- 此外，再将“帖子”集合添加到“博客”类，以形成“博客”和“帖子”之间的另一层关系

```
public virtual List<Post> Posts { get; set; }
```

使用 Add-Migration 命令使 Code First 迁移提供对迁移的最佳猜测 我们将调用此迁移 AddPostClass。

- 在包管理器控制台中运行 Add-Migration AddPostClass 命令。

Code First 迁移出色的构建了这些更改，但我们可能还需要做出一些更改：

1. 首先，将唯一索引添加到 Posts.Title 列（添加在以下代码的 22 和 29 行）。
2. 同时添加不可为 NULL 的 Blogs.Rating 列。如果表中存在任何现有数据，则会为数据分配新列数据类型的 CLR 默认值（分级是整数，因此将为 0）。但我们要将默认值指定为 3，以便“博客”表中的现有行以合适的分级开始。（可以看到以下代码的第 24 行指定的默认值）

```

namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPostClass : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Posts",
                c => new
                {
                    PostId = c.Int(nullable: false, identity: true),
                    Title = c.String(maxLength: 200),
                    Content = c.String(),
                    BlogId = c.Int(nullable: false),
                })
                .PrimaryKey(t => t.PostId)
                .ForeignKey("dbo.Blogs", t => t.BlogId, cascadeDelete: true)
                .Index(t => t.BlogId)
                .Index(p => p.Title, unique: true);

            AddColumn("dbo.Blogs", "Rating", c => c.Int(nullable: false, defaultValue: 3));
        }

        public override void Down()
        {
            DropIndex("dbo.Posts", new[] { "Title" });
            DropIndex("dbo.Posts", new[] { "BlogId" });
            DropForeignKey("dbo.Posts", "BlogId", "dbo.Blogs");
            DropColumn("dbo.Blogs", "Rating");
            DropTable("dbo.Posts");
        }
    }
}

```

已编辑的迁移准备就绪，所以我们使用 Update-Database 来更新数据库。这次指定 -Verbose 标志，以便可以看到 Code First 迁移正在运行的 SQL。

- 在包管理器控制台中运行 Update-Database -Verbose 命令。

数据移动/自定义 SQL

到目前为止，我们已经介绍了不更改或移动任何数据的迁移操作，现在来看看需要移动数据的操作。目前还没有对数据移动的原生支持，但我们可以从脚本中的任何位置运行一些任意 SQL 命令。

- 将 Post.Abstract 属性添加到模型中。稍后，我们将使用“内容”列开头的一些文本预填充现有帖子的“摘要”。

```
public string Abstract { get; set; }
```

使用 Add-Migration 命令使 Code First 迁移提供对迁移的最佳猜测

- 在包管理器控制台中运行 Add-Migration AddPostAbstract 命令。
- 生成的迁移会处理架构更改，但我们还是希望使用每个帖子内容的前 100 个字符预填充“摘要”列。要执行此操作，可在添加列之后下拉到 SQL 并运行 UPDATE 语句。（添加在以下代码的第 12 行中）

```

namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPostAbstract : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Posts", "Abstract", c => c.String());

            Sql("UPDATE dbo.Posts SET Abstract = LEFT(Content, 100) WHERE Abstract IS NULL");
        }

        public override void Down()
        {
            DropColumn("dbo.Posts", "Abstract");
        }
    }
}

```

已编辑的迁移一切正常，所以我们可使用 Update-Database 来更新数据库。我们将指定 –Verbose 标志，以便可以看到针对数据库运行的 SQL。

- 在包管理器控制台中运行 Update-Database –Verbose 命令。

迁移到特定版本(包括降级)

到目前为止，我们一直在升级到最新迁移，但用户有时可能希望升级/降级到特定迁移。

假设想在运行 AddBlogUrl 迁移后将数据库迁移到其之前的状态。此时可使用 –TargetMigration 切换为降级到此迁移。

- 在包管理器控制台中运行 Update-Database –TargetMigration: AddBlogUrl 命令。

此命令将为 AddBlogAbstract 和 AddPostClass 迁移运行 Down 脚本。

如果想要一直回退到空数据库，可使用 Update-Database –TargetMigration: \$InitialDatabase 命令。

获取 SQL 脚本

如果另一位开发人员希望在其计算机上进行这些更改，则只需在我们将更改签入源代码管理之后进行同步即可。在获得我们的新迁移后，他们只需运行 Update-database 命令即可在本地应用更改。但是，如果想将这些更改推送到测试服务器以及最终的产品，则可能需要一个可以传递给 DBA 的 SQL 脚本。

- 运行 Update-Database 命令，但是这次需指定 –Script 标志，以便将更改写入脚本，而不是应用更改。我们还将指定要为其生成脚本的源和目标迁移。我们希望脚本从空数据库 (\$InitialDatabase) 转为最新版本(迁移 AddPostAbstract)。如果未指定目标迁移，迁移将使用最新的迁移作为目标。如果未指定源迁移，迁移将使用数据库的当前状态。
- 在包管理器控制台中运行 Update-Database -Script -SourceMigration: \$InitialDatabase -TargetMigration: AddPostAbstract 命令

Code First 迁移将运行迁移管道，但并非是应用更改，而是将更改写入到 .sql 文件。生成脚本后，将在 Visual Studio 中打开，以供查看或保存。

生成幂等脚本

从 EF6 开始，如果指定 –SourceMigration \$InitialDatabase，则生成的脚本将为“幂等”。幂等脚本可以将当前任何版本的数据库升级到最新版本(或使用 –TargetMigration 升级到指定版本)。生成的脚本包括检查 __MigrationsHistory 表的逻辑，并且仅应用以前未应用的更改。

应用程序启动时自动升级 (MigrateDatabaseToLatestVersion 初始值设定项)

如果要部署应用程序，则可能希望应用程序在启动时自动升级数据库（通过应用各种挂起的迁移）。可以通过注册 MigrateDatabaseToLatestVersion 数据库初始值设定项来执行此操作。数据库初始值设定项仅包含一些用于确保正确设置数据库的逻辑。第一次在应用程序进程中使用上下文时，会运行此逻辑（AppDomain）。

如下所示，可以更新 Program.cs 文件，以在使用上下文（第 14 行）之前，为 BlogContext 设置 MigrateDatabaseToLatestVersion 初始化值设定项。请注意，还需要为 System.Data.Entity 命名空间（第 5 行）添加 using 语句。

创建此初始值设定项的实例时，需要指定上下文类型（BlogContext）和迁移配置（配置）- 迁移配置是启用迁移时添加到“迁移”文件夹的类。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using MigrationsDemo.Migrations;

namespace MigrationsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<BlogContext, Configuration>());

            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog" });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

现在，每当应用程序运行时，它首先会检查其目标数据库是否为最新，如果不是，则会应用各种挂起的迁移。

自动 Code First 迁移

2020/3/11 ·

自动迁移使你可以使用 Code First 迁移，而不会在项目中为你所做的每个更改提供代码文件。并非所有更改都可以自动应用-例如，列重命名需要使用基于代码的迁移。

NOTE

本文假设你知道如何在基本方案中使用 Code First 迁移。如果没有，则需要先阅读[Code First 迁移](#)，然后再继续。

团队环境建议

你可以点播自动和基于代码的迁移，但不建议在团队开发方案中使用。如果你是使用源代码管理的开发人员团队的成员，则应使用纯粹自动迁移或纯粹基于代码的迁移。考虑到自动迁移的限制，我们建议在团队环境中使用基于代码的迁移。

生成初始模型和数据库

开始使用迁移之前，需要会用到项目和 Code First 模型。对于此演练，我们将使用规范的“博客”和“帖子”模型。

- 创建新的MigrationsAutomaticDemo控制台应用程序
- 将最新版本的 EntityFramework NuGet 包添加到项目中
 - “工具”–“库包管理器”–>“包管理器控制台”>
 - 运行 Install-Package EntityFramework 命令
- 添加 Model.cs 文件，其代码如下所示。此代码定义了构成域模型的单个“博客”类和 EF Code First 上下文 BlogContext 类

```
using System.Data.Entity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.Infrastructure;

namespace MigrationsAutomaticDemo
{
    public class BlogContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

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

- 现在我们拥有一个模型，可用它执行数据访问操作。更新 Program.cs 文件，其代码如下所示。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

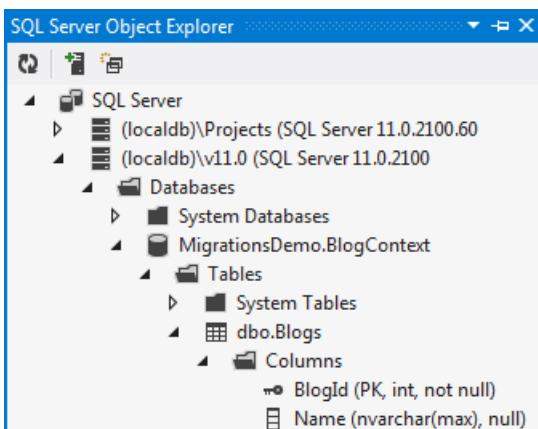
namespace MigrationsAutomaticDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog " });
                db.SaveChanges();

                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(blog.Name);
                }
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

- 运行你的应用程序，你将看到已为你创建**MigrationsAutomaticCodeDemo. BlogContext**数据库。



启用迁移

现可对模型进行更多更改。

- 将 Url 属性引入“博客”类。

```
public string Url { get; set; }
```

如果您要再次运行该应用程序，将会出现一个 InvalidOperationException，指出支持 “BlogContext” 上下文的模型在创建数据库后发生了更改。请考虑使用 Code First 迁移更新数据库 (<http://go.microsoft.com/fwlink/?LinkId=238269>)。

如异常情况所述，可开始使用 Code First 迁移。由于我们要使用自动迁移，因此，我们将指定 –EnableAutomaticMigrations 开关。

- 在包管理器控制台中运行 "启用-迁移– EnableAutomaticMigrations " 命令。此命令已向项目添加了 "迁移" 文件夹。此新文件夹包含一个文件：

- 配置类。此类允许配置迁移对上下文的行为方式。对于此演练，将只使用默认配置。由于项目中只有一个 Code First 上下文，因此 Enable-Migrations 已自动填充此配置适用的上下文类型。

第一次自动迁移

Code First 迁移具有两个需要用户了解的主要命令。

- Add-Migration 将基于自上次迁移创建以来对模型所做的更改来构建下一次迁移
- Update-Database 将对数据库应用任意挂起的迁移

我们将避免使用添加迁移(除非我们确实需要)，并专注于让 Code First 迁移自动计算和应用更改。让我们使用更新数据库来获取 Code First 迁移将对模型(新的博客 Url 属性)的更改推送到数据库。

- 在 Package Manager Console 中运行 "更新数据库" 命令。

MigrationsAutomaticDemo BlogContext 数据库现在已更新为包含博客表中的Url列。

第二次自动迁移

接下来，让我们进行其他更改，并让 Code First 迁移自动将更改推送到数据库。

- 同时添加一个新的“帖子”类

```
public class Post
{
    public int PostId { get; set; }
    [MaxLength(200)]
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

- 此外，再将“帖子”集合添加到“博客”类，以形成“博客”和“帖子”之间的另一层关系

```
public virtual List<Post> Posts { get; set; }
```

现在使用更新数据库使数据库保持最新状态。这次指定 -Verbose 标志，以便可以看到 Code First 迁移正在运行的 SQL。

- 在包管理器控制台中运行 Update-Database -Verbose 命令。

添加基于代码的迁移

现在，让我们看看我们可能要对使用基于代码的迁移。

- 让我们向博客类添加评级属性

```
public int Rating { get; set; }
```

只需运行 "更新数据库" 即可将这些更改推送到数据库。但是，我们添加了不可为 null 的博客。“分级”列，如果表中存在任何现有数据，则会向其分配新列的数据类型的 CLR 默认值(评级为整数，以便为0)。但我们要将默认值指

定为 3，以便“博客”表中的现有行以合适的分级开始。让我们使用“添加-迁移”命令将此更改写入到基于代码的迁移，以便可以对其进行编辑。“添加-迁移”命令使我们可以为这些迁移命名，只需调用我们的AddBlogRating即可。

- 在 Package Manager Console 中运行 Add-迁移 AddBlogRating 命令。
- 在“迁移”文件夹中，我们现在已有一个新的AddBlogRating迁移。迁移文件名预先修复了时间戳，以帮助进行排序。让我们编辑生成的代码，为博客指定默认值3（在下面的代码中为第10行）

迁移还具有捕获某些元数据的代码隐藏文件。此元数据将允许 Code First 迁移复制在此基于代码的迁移之前执行的自动迁移。如果另一位开发人员想要运行迁移或部署应用程序，这一点非常重要。

```
namespace MigrationsAutomaticDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddBlogRating : DbMigration
    {
        public override void Up()
        {
            AddColumn("Blogs", "Rating", c => c.Int(nullable: false, defaultValue: 3));
        }

        public override void Down()
        {
            DropColumn("Blogs", "Rating");
        }
    }
}
```

已编辑的迁移一切正常，所以我们可使用 Update-Database 来更新数据库。

- 在 Package Manager Console 中运行“更新数据库”命令。

返回到自动迁移

现在，我们可以免费切换回自动迁移，以便进行更简单的更改。Code First 迁移将按照在每个基于代码的迁移的代码隐藏文件中存储的元数据，以正确的顺序执行自动和基于代码的迁移。

- 让我们向模型中添加一个 Post.Abstract 属性

```
public string Abstract { get; set; }
```

现在，我们可以使用更新数据库来获取 Code First 迁移，使用自动迁移将此更改推送到数据库。

- 在 Package Manager Console 中运行“更新数据库”命令。

摘要

在本演练中，您学习了如何使用自动迁移将模型更改推送到数据库。还了解了在需要更多控制时，如何在自动迁移之间基架和运行基于代码的迁移。

使用现有数据库 Code First 迁移

2020/3/11 ·

NOTE

Ef 4.3 在实体框架4.1 中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

本文介绍如何将 Code First 迁移与现有数据库(不是由实体框架创建的数据库)结合使用。

NOTE

本文假设你知道如何在基本方案中使用 Code First 迁移。如果没有，则需要先阅读[Code First 迁移](#)，然后再继续。

屏幕广播

如果你想要观看 screencast 而不是阅读本文，以下两个视频会涵盖与本文相同的内容。

视频一：“迁移-在后台”

[此 screencast](#)介绍了迁移如何跟踪并使用有关模型的信息来检测模型更改。

视频两个：“迁移-现有数据库”

基于前一视频中的概念生成，[此 screencast](#)介绍了如何启用和使用现有数据库的迁移。

步骤1：创建模型

第一步是创建一个面向现有数据库的 Code First 模型。“[Code First 现有数据库](#)”主题提供有关如何执行此操作的详细指导。

NOTE

在对模型进行任何需要更改数据库架构的更改之前，请务必遵循本主题中的其余步骤。以下步骤要求模型与数据库架构保持同步。

步骤2：启用迁移

下一步是启用迁移。可以通过在 Package Manager Console 中运行 “启用-迁移” 命令来执行此操作。

此命令将在你的解决方案中创建一个名为 “迁移” 的文件夹，并在其中放置一个名为 “配置” 的类。配置类是你为应用程序配置迁移的位置，你可以在[Code First 迁移](#)主题中找到有关它的详细信息。

步骤3：添加初始迁移

创建迁移并将迁移应用到本地数据库后，您可能还需要将这些更改应用到其他数据库。例如，您的本地数据库可能是一个测试数据库，您可能最终还需要对生产数据库和/或其他开发人员测试数据库应用所做的更改。此步骤有两个选项，应选择的选项取决于任何其他数据库的架构是否为空，或当前是否与本地数据库的架构匹配。

- 选项一：使用现有架构作为起始点。如果将来将应用迁移到的其他数据库将具有与本地数据库当前具有的架构相同的架构，则应使用此方法。例如，如果本地测试数据库当前与生产数据库的 v1 相匹配，则可以使用此项，稍后会应用这些迁移，将生产数据库更新为 v2。

- 选项2: 使用空数据库作为起始点。如果将来将应用迁移的其他数据库为空(或尚不存在), 则应使用此方法。例如, 如果使用测试数据库开始开发应用程序, 但不使用迁移, 则可以使用此示例, 以后要从头开始创建生产数据库。

选项一: 使用现有架构作为起始点

Code First 迁移使用最新迁移中存储的模型的快照来检测对模型所做的更改(可在[团队环境中 Code First 迁移](#)找到有关此内容的详细信息)。由于我们将假设数据库已具有当前模型的架构, 因此, 我们将生成一个将当前模型作为快照的空(无操作)迁移。

1. 在 Package Manager Console 中运行Add-迁移 InitialCreate – IgnoreChanges命令。这将创建一个空迁移, 并将当前模型作为快照。
2. 在 Package Manager Console 中运行 "更新数据库" 命令。这会将 InitialCreate 迁移应用到数据库。由于实际迁移并不包含任何更改, 因此只需将一行添加到 __MigrationsHistory 表, 指出已应用此迁移。

选项2: 使用空数据库作为起始点

在此方案中, 我们需要迁移才能从头开始创建整个数据库(包括本地数据库中已存在的表)。我们将生成一个 InitialCreate 迁移, 其中包含用于创建现有架构的逻辑。然后, 将使现有的数据库看起来像此迁移已应用。

1. 在 Package Manager Console 中运行Add-迁移 InitialCreate命令。这将创建一个迁移来创建现有架构。
2. 注释掉新创建的迁移的 Up 方法中的所有代码。这样, 我们就可以 "应用" 将迁移到本地数据库, 而无需尝试重新创建所有表, 等等。
3. 在 Package Manager Console 中运行 "更新数据库" 命令。这会将 InitialCreate 迁移应用到数据库。由于实际的迁移不包含任何更改(因为我们暂时注释它们), 因此它只是将一行添加到 __MigrationsHistory 表, 指示已应用此迁移。
4. 取消注释 Up 方法中的代码。这意味着, 将此迁移应用于未来的数据库时, 迁移将创建本地数据库中已经存在的架构。

注意事项

对现有数据库使用迁移时, 需要注意几个事项。

默认/计算的名称可能与现有架构不匹配

迁移时, 迁移会显式指定列和表的名称基架。但是, 在应用迁移时, 迁移会为其他数据库对象计算默认名称。这包括索引和外键约束。当以现有架构为目标时, 这些计算的名称可能与数据库中实际存在的名称不匹配。

下面是需要注意的一些示例:

如果使用步骤3中的 "选项一: 使用现有架构作为起点":

- 如果模型中的未来更改需要更改或删除一个名为不同的数据库对象, 则需要修改基架迁移来指定正确的名称。迁移 API 有一个可选的 Name 参数, 可让你执行此操作。例如, 你的现有架构可能有一个带有 BlogId 外键列的 Post 表, 该表的索引名为 IndexFk_BlogId。但是, 默认情况下, 迁移应将此索引命名为 IX_BlogId。如果对模型进行更改, 导致删除此索引, 则需要修改基架 DropIndex 调用以指定 IndexFk_BlogId 名称。

如果使用步骤3中的 "Option 2: 使用空数据库作为起始点":

- 尝试对本地数据库运行初始迁移的关闭方法(即, 恢复为空数据库)可能会失败, 因为迁移将尝试使用错误的名称删除索引和外键约束。这将会影响本地数据库, 因为将使用初始迁移的 Up 方法从头开始创建其他数据库。如果要将现有的本地数据库降级到空状态, 可以通过删除数据库或删除所有表, 手动执行此操作, 这是最简单的方法。在此初始降级后, 将用默认名称重新创建所有数据库对象, 因此, 此问题不会再次出现。
- 如果您的模型中的未来更改需要更改或删除一个名为不同的数据库对象, 则这对您的现有本地数据库不起作用, 因为这些名称与默认值不匹配。但是, 它将适用于 "从头开始创建" 的数据库, 因为它们将使用迁移选择的默认名称。您可以在本地现有数据库上手动进行这些更改, 也可以考虑让迁移从头开始重新创建数据库, 因为它将在其他计算机上进行。
- 使用初始迁移的 Up 方法创建的数据库可能与本地数据库略有不同, 因为将使用计算的索引和外键约束的默认

名称。你还可能最终会获得额外的索引，因为迁移将默认在外键列上创建索引-这可能不是你的原始本地数据库中的情况。

不是所有数据库对象都是在模型中表示的

不属于您的模型的数据库对象将不会被迁移处理。这可能包括视图、存储过程、权限、不是模型一部分的表、附加索引等。

下面是需要注意的一些示例：

- 无论你在 "步骤 3" 中选择哪个选项，如果将来对模型进行更改，则需要更改或删除这些其他对象。迁移将不知道进行这些更改。例如，如果您删除了一个具有其他索引的列，迁移将不知道删除该索引。你将需要手动将其添加到基架迁移。
- 如果使用了 "Option 2: 使用空数据库作为起始点"，则将不会通过初始迁移的 "向上" 方法创建这些附加的对象。如果需要，可以修改向上和向下方法来处理这些额外的对象。对于迁移 API 中不是本机支持的对象（如视图），可以使用[Sql](#)方法运行原始 Sql 来创建/删除它们。

自定义迁移历史记录表

2020/3/11 ·

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

NOTE

本文假设你知道如何在基本方案中使用 Code First 迁移。如果没有，则需要先阅读[Code First 迁移](#)，然后再继续。

什么是迁移历史记录表？

迁移历史记录表是 Code First 迁移用来存储应用于数据库的迁移的详细信息的表。默认情况下，数据库中的表名称 `_MigrationHistory`，并在应用首次迁移到数据库时创建。在实体框架5中，如果应用程序使用 Microsoft Sql Server 数据库，则此表为系统表。这在实体框架6中已更改，并且迁移历史记录表不再标记为系统表。

为什么自定义迁移历史记录表？

迁移历史记录表应仅由 Code First 迁移使用，并可以手动更改，从而中断迁移。但有时，默认配置不合适，需要对表进行自定义，例如：

- 需要更改列的名称和/或 facet，以启用3个rd方迁移提供程序
- 要更改表的名称
- 需要为 `_MigrationHistory` 表使用非默认架构
- 需要为特定版本的上下文存储其他数据，因此需要向表中添加其他列

预防词

更改迁移历史记录表的功能非常强大，但需小心不要过度编写。EF 运行时当前不检查自定义迁移历史记录表是否与运行时兼容。如果不是，你的应用程序可能会在运行时中断或以不可预知的方式运行。如果对每个数据库使用多个上下文，这一点更重要，在这种情况下，多个上下文可以使用相同的迁移历史记录表来存储有关迁移的信息。

如何自定义迁移历史记录表？

在开始之前，你需要知道只能在应用首次迁移之前自定义迁移历史记录表。现在，到代码。

首先，你将需要创建一个派生自 `HistoryContext` 类的类，`.HistoryContext` 类派生自 `DbContext` 类，因此配置迁移历史记录表非常类似于配置 EF 模型与 Fluent API。只需重写 `OnModelCreating` 方法并使用 Fluent API 来配置该表。

NOTE

通常，在配置 EF 模型时，无需调用 `base.OnModelCreating()` 来重写 `OnModelCreating` 方法，因为 `OnModelCreating()` 的主体为空。配置迁移历史记录表时不会出现这种情况。在这种情况下，首先要在 `OnModelCreating()` 重写中执行此操作。`OnModelCreating()`。这会将迁移历史记录表配置为默认方法，然后在重写方法中进行调整。

假设你想要重命名迁移历史记录表，并将其放入名为 "admin" 的自定义架构。此外，DBA 需要将 `MigrationId` 列重

命名为`_ID`的迁移。可以通过创建以下派生自`HistoryContext`的类来实现此目的：

```
using System.Data.Common;
using System.Data.Entity;
using System.Data.Entity.Migrations.History;

namespace CustomizableMigrationsHistoryTableSample
{
    public class MyHistoryContext : HistoryContext
    {
        public MyHistoryContext(DbConnection dbConnection, string defaultSchema)
            : base(dbConnection, defaultSchema)
        {
        }

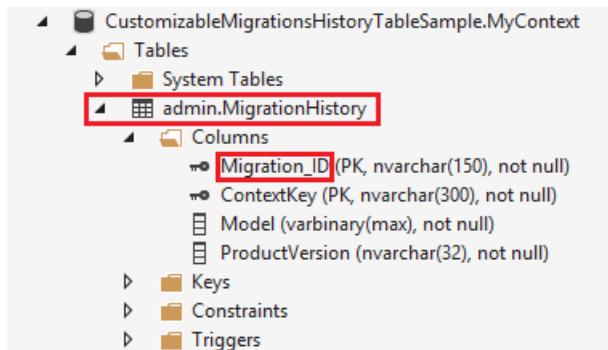
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<HistoryRow>().ToTable(tableName: "MigrationHistory", schemaName: "admin");
            modelBuilder.Entity<HistoryRow>().Property(p => p.MigrationId).HasColumnName("Migration_ID");
        }
    }
}
```

自定义`HistoryContext`准备就绪后，你需要通过[基于代码的配置](#)注册EF来了解它：

```
using System.Data.Entity;

namespace CustomizableMigrationsHistoryTableSample
{
    public class ModelConfiguration : DbConfiguration
    {
        public ModelConfiguration()
        {
            this.SetHistoryContext("System.Data.SqlClient",
                (connection, defaultSchema) => new MyHistoryContext(connection, defaultSchema));
        }
    }
}
```

就这么多了。现在，你可以开始使用包管理器控制台，启用-迁移，添加-迁移，最后更新-数据库。这会导致将迁移历史记录表添加到数据库，并根据你在`HistoryContext`派生类中指定的详细信息进行配置。



使用 debug.exe

2020/3/11 •

Code First 迁移可用于从 visual studio 内部更新数据库，但也可以通过命令行工具 debug.exe 来执行。此页将简要介绍如何使用 debug.exe 来对数据库执行迁移。

NOTE

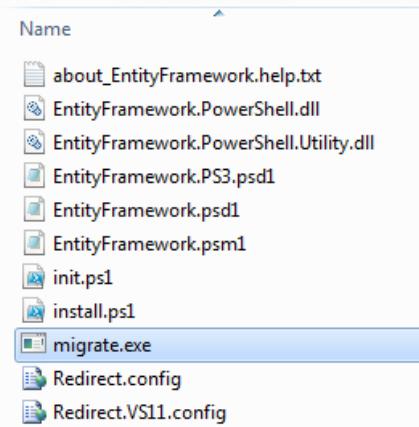
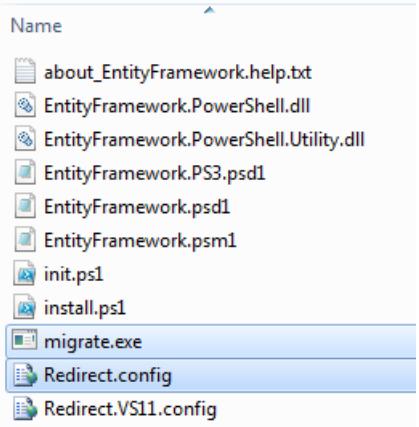
本文假设你知道如何在基本方案中使用 Code First 迁移。如果没有，则需要先阅读[Code First 迁移](#)，然后再继续。

Copy debug.exe

当你安装实体框架时，将在下载包的 tools 文件夹内使用 NuGet 迁移。在 <项目文件夹中>\包\EntityFramework。<版本>\工具

迁移 .exe 后，需要将其复制到包含迁移的程序集所在的位置。

如果你的应用程序面向 .NET 4，而不是4.5，则你需要将重定向配置复制到该位置，并将其重命名为debug.exe。这是因为，debug.exe 会获得正确的绑定重定向，以便能够找到实体框架程序集。

.NET 4.5	.NET 4.0
	

NOTE

debug.exe 不支持 x64 程序集。

将 nodejs 移动到正确的文件夹后，应该能够使用它来对数据库执行迁移。此实用程序的目的是执行迁移。它无法生成迁移或创建 SQL 脚本。

查看选项

```
Migrate.exe /?
```

以上将显示与此实用程序关联的帮助页，请注意，你需要在运行的同一位置中具有 EntityFramework，才能使其正常运行。

迁移到最新迁移

```
Migrate.exe MyMvcApplication.dll /startupConfigurationFile="..\web.config"
```

运行 "迁移" 时，唯一的必需参数是程序集，该程序集是包含您尝试运行的迁移的程序集，但如果您未指定配置文件，它将使用所有基于约定的设置。

迁移到特定迁移

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /targetMigration="AddTitle"
```

如果要运行到特定迁移的迁移，则可以指定迁移的名称。这会根据需要运行所有以前的迁移，直到获取所指定的迁移。

指定工作目录

```
Migrate.exe MyApp.exe /startupConfigurationFile="MyApp.exe.config" /startupDirectory="c:\\MyApp"
```

如果程序集具有依赖项或读取相对于工作目录的文件，则需要设置 startupDirectory。

指定要使用的迁移配置

```
Migrate.exe MyAssembly CustomConfig /startupConfigurationFile="..\web.config"
```

如果有多个迁移配置类，继承自 DbMigrationConfiguration 的类，则需要指定将用于此执行的类。这是通过提供可选的第二个参数(不带上述开关)来指定的。

提供连接字符串

```
Migrate.exe BlogDemo.dll /connectionString="Data Source=localhost;Initial Catalog=BlogDemo;Integrated Security=SSPI" /connectionProviderName="System.Data.SqlClient"
```

如果要在命令行中指定连接字符串，则还必须提供提供程序名称。如果未指定提供程序名称，将导致异常。

常见问题

错误	原因
未经处理的异常: FileLoadException: 无法加载文件或程序集 "EntityFramework, Version = 5.0.0.0, Culture = 中立, PublicKeyToken = b77a5c561934e089" 或其依赖项之一。找到的程序集清单定义与程序集引用不匹配。(异常来自 HRESULT:0x80131040)	这通常意味着您在运行 .NET 4 应用程序时没有重定向的 .config 文件。需要将重定向配置复制到与 debug.exe 相同的位置，并将其重命名为 "迁移"。
未经处理的异常: FileLoadException: 无法加载文件或程序集 "EntityFramework, Version = 4.4.0.0, Culture = 中立, PublicKeyToken = b77a5c561934e089" 或其依赖项之一。找到的程序集清单定义与程序集引用不匹配。(异常来自 HRESULT:0x80131040)	此异常表示你正在运行 .NET 4.5 应用程序，并将重定向配置复制到迁移 .exe 位置。如果应用是 .NET 4.5，则不需要在内部重定向配置文件。删除迁移文件。

错误:无法更新数据库以匹配当前模型,因为存在挂起的更改,并且禁用了自动迁移。将挂起的模型更改写入到基于代码的迁移或启用自动迁移。将 DbMigrationsConfiguration 设置为 true 以后用自动迁移。	如果尚未创建迁移来处理对模型所做的更改,并且数据库与模型不匹配,则会发生此错误。将属性添加到模型类后,如果不创建迁移来升级数据库,就会出现这种情况。
错误:未解析成员 "ToolingFacade + UpdateRunner, EntityFramework, Version = 5.0.0.0, Culture = 中立, PublicKeyToken = b77a5c561934e089" 的类型。	此错误的原因可能是指定了错误的启动目录。这必须是 debug.exe 的位置
未经处理的异常: NullReferenceException:对象引用未设置为对象的实例。 在 System.web. Main (String [] args) 上。	这可能是由于未为你使用的方案指定必需参数引起的。例如,在不指定提供程序名称的情况下指定连接字符串。
错误:在程序集 "Classlibrary1.chainone" 中找到了多个迁移配置类型。指定要使用的名称。	作为错误状态,在给定的程序集中有多个配置类。必须使用 /configurationType 开关来指定要使用的。
错误:无法加载文件或程序集 "<assemblyName>" 或其依赖项之一。给定的程序集名称或基本代码无效。(异常来自 HRESULT:0x80131047)	这可能是由于不正确地指定程序集名称或
错误:无法加载文件或程序集 "<assemblyName>" 或其依赖项之一。试图加载的程序的格式不正确。	如果尝试对 x64 应用程序运行迁移,则会发生这种情况。EF 5.0 和更低的将仅适用于 x86。

团队环境中的 Code First 迁移

2020/3/11 ·

NOTE

本文假设你知道如何在基本方案中使用 Code First 迁移。如果没有，则需要先阅读[Code First 迁移](#)，然后再继续。

抓住咖啡，你需要阅读这篇文章

团队环境中的问题主要涉及两个开发人员在其本地基本代码中生成迁移时的合并。虽然解决这些问题的步骤非常简单，但需要您对迁移的工作方式有深刻的理解。请不要直接跳到结尾—请花时间阅读整篇文章，以确保成功。

一些一般准则

在深入探讨如何管理由多个开发人员生成的合并迁移之前，下面是一些用于设置成功的一般准则。

每个团队成员都应具有一个本地开发数据库

迁移使用 `_MigrationsHistory` 表来存储已应用到数据库的迁移。如果有多个开发人员在尝试面向同一数据库时生成不同的迁移（因而共享一个 `_MigrationsHistory` 表），则迁移将会非常困惑。

当然，如果您的团队成员不生成迁移，则不会有任何问题与中央开发数据库共享。

避免自动迁移

最后一行是，自动迁移最初在团队环境中看起来很好，但实际上它们不起作用。如果希望了解原因，请继续阅读—否则，可以跳到下一部分。

自动迁移使你能够更新数据库架构以匹配当前模型，而无需生成代码文件（基于代码的迁移）。如果你只使用了这些迁移，并且从未生成任何基于代码的迁移，则自动迁移将在团队环境中正常工作。问题在于自动迁移是有限的，并且不处理多个操作—属性/列重命名、将数据移动到另一个表等。若要处理这些情况，您最终会生成基于代码的迁移（和编辑基架代码），这些迁移在自动迁移处理的更改之间混合使用。这使得在两个开发人员签入迁移时，无法合并更改。

屏幕广播

如果你想要观看 screencast 而不是阅读本文，以下两个视频会涵盖与本文相同的内容。

视频一：“迁移-在后台”

此 screencast 介绍了迁移如何跟踪并使用有关模型的信息来检测模型更改。

视频两个：“迁移-团队环境”

根据前一视频中的概念构建，此 screencast 涵盖了团队环境中出现的问题以及如何解决这些问题。

了解迁移的工作方式

在团队环境中成功使用迁移的关键是要了解迁移如何跟踪并使用有关模型的信息来检测模型更改。

第一次迁移

将首次迁移添加到项目时，会在程序包管理器控制台中首先运行类似于添加迁移的操作。此命令执行的高级步骤如下图所示。

PM> Add-Migration First

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

1 Build current model from code



2 Calculate required database changes

EdmModelDiffer

MigrationCodeGenerator

3 Generate code to apply changes

4 Generated files added to project

First.cs

CreateTable("dbo.Blogs")

First.resx

Snapshot of model at time of generation is stored as

an embedded resource.



当前模型是从您的代码(1)计算得到的。然后，模型将计算所需的数据库对象(2)-由于这是第一次迁移模型，因此模型不同只是使用空模型进行比较。所需的更改会传递到代码生成器，以生成所需的迁移代码(3)，然后将其添加到你的 Visual Studio 解决方案(4)。

除了存储在主代码文件中的实际迁移代码外，迁移还会生成一些附加的代码隐藏文件。这些文件是迁移使用的元数据，而不是你应该编辑的内容。其中一项文件是资源文件(.resx)，其中包含在生成迁移时模型的快照。下一步将介绍如何使用此方法。

此时，你可能会运行 "更新数据库" 以将更改应用到数据库，然后再开始实现应用程序的其他区域。

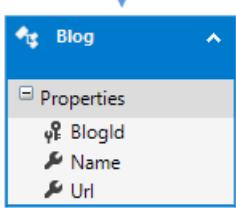
后续迁移

稍后返回并对模型进行一些更改-在本示例中，我们会将Url属性添加到博客。然后，将发出一个命令(如添加迁移 AddUrl)基架迁移，以应用相应的数据库更改。此命令执行的高级步骤如下图所示。

PM> Add-Migration AddUrl

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
}
```

1 Build current model from code



2 Get previous model from last migration

First.cs

CreateTable("dbo.Blogs")

First.resx

Snapshot of model at time of generation is stored as

an embedded resource.

3 Calculate required database changes

EdmModelDiffer

MigrationCodeGenerator

4 Generate code to apply changes

5 Generated files added to project

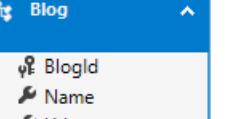
AddUrl.cs

AddColumn("dbo.Blogs", "Rat

AddUrl.resx

Snapshot of model at time of generation is stored as

an embedded resource.



与上一次一样，当前模型是从代码(1)计算而来的。但是，这一次存在现有的迁移，因此从最新迁移(2)中检索上一个模型。这两个模型将 diffed，以查找所需的数据库更改(3)，然后该进程将像以前一样完成。

此过程用于添加到项目中的任何其他迁移。

为什么要干扰模型快照？

您可能想知道 EF 为什么要麻烦模型快照—为什么不只是查看数据库。如果是这样，请继续阅读。如果你不感兴趣，则可以跳过此部分。

EF 保留模型快照的原因有很多：

- 它允许您的数据库与 EF 模型的偏差。可以直接在数据库中进行这些更改，也可以更改迁移中的基架代码以进行更改。下面是其中几个示例：
 - 您希望向一个或多个表中添加插入的和更新的列，但不希望在 EF 模型中包含这些列。如果迁移过程中查看了数据库，则每次基架迁移时，它都会不断尝试删除这些列。使用模型快照，EF 只会检测到对模型的合法更改。
 - 要更改用于更新的存储过程的正文，以包含一些日志记录。如果迁移从数据库中查看此存储过程，它将继续尝试并将其重置回 EF 需要的定义。通过使用模型快照，在 EF 模型中更改过程的形状时，EF 只会基架代码来更改存储过程。
 - 这些相同的原则适用于添加额外的索引，包括数据库中的额外表、将 EF 映射到表中的数据库视图等。
- EF 模型只包含数据库的形状。整个模型允许迁移查看有关模型中的属性和类的信息，以及它们如何映射到这些列和表。此信息允许在基架的代码中更智能地迁移。例如，如果您更改属性映射到迁移的列的名称，则可以通过查看它是相同的属性来检测重命名，如果只有数据库架构，则无法执行此操作。

导致团队环境中出现问题的原因

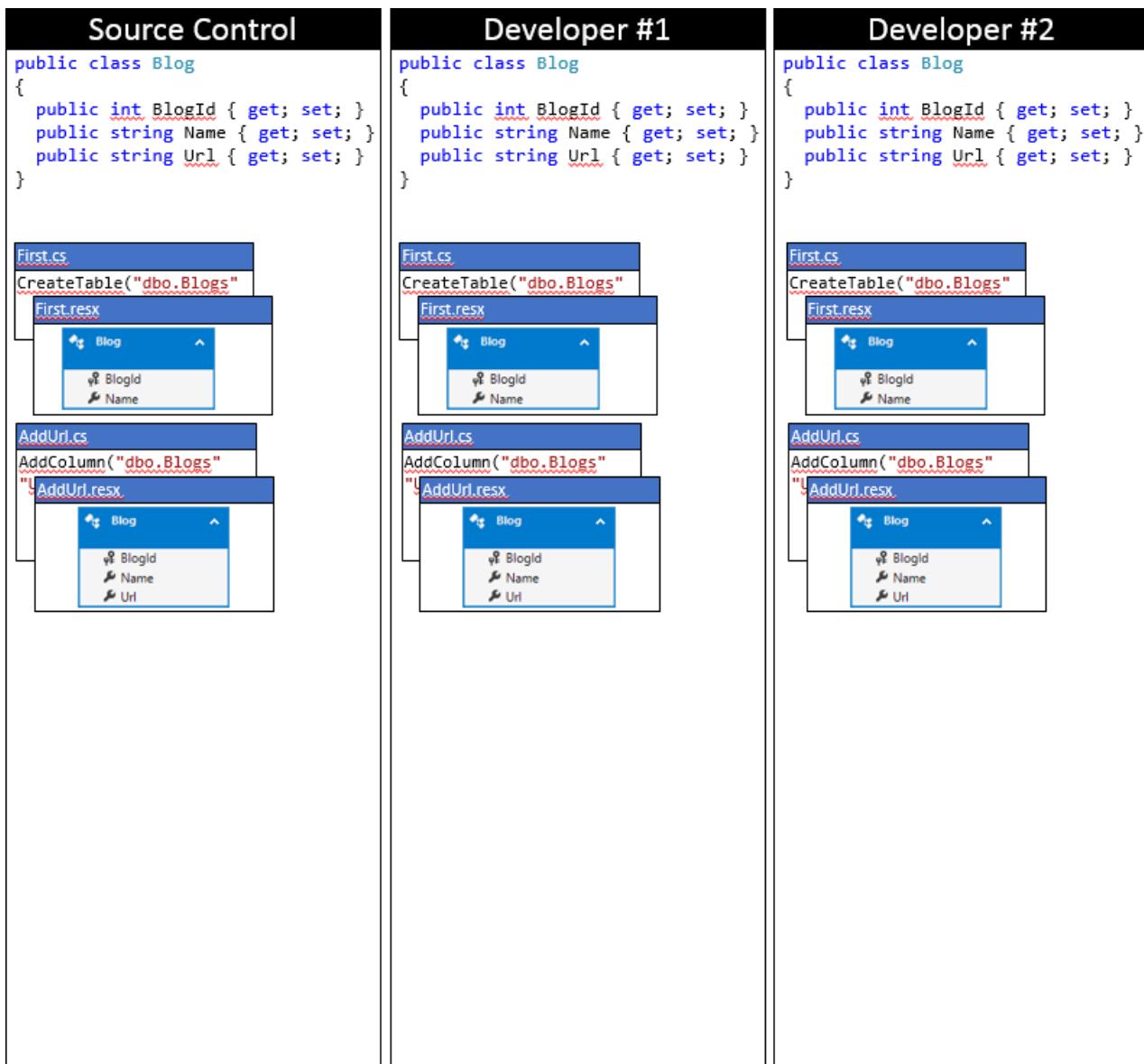
当你是处理应用程序的单个开发人员时，上一部分介绍的工作流非常有用。如果您是更改模型的唯一人员，它在团队环境中也能正常工作。在这种情况下，你可以进行模型更改，生成迁移并将它们提交到源控件。其他开发人员可以同步您的更改并运行更新数据库，以应用架构更改。

当你有多个开发人员更改 EF 模型并同时提交到源代码管理中时，就会出现问题。EF 缺乏哪一种方法是将本地迁移与其他开发人员自上次同步后已提交到源代码管理的迁移合并在一起。

合并冲突的示例

首先，让我们看一看此类合并冲突的具体示例。我们会继续学习前面所述的示例。作为起点，假设先前部分中的更改已由原始开发人员签入。我们将在两个开发人员更改代码库时进行跟踪。

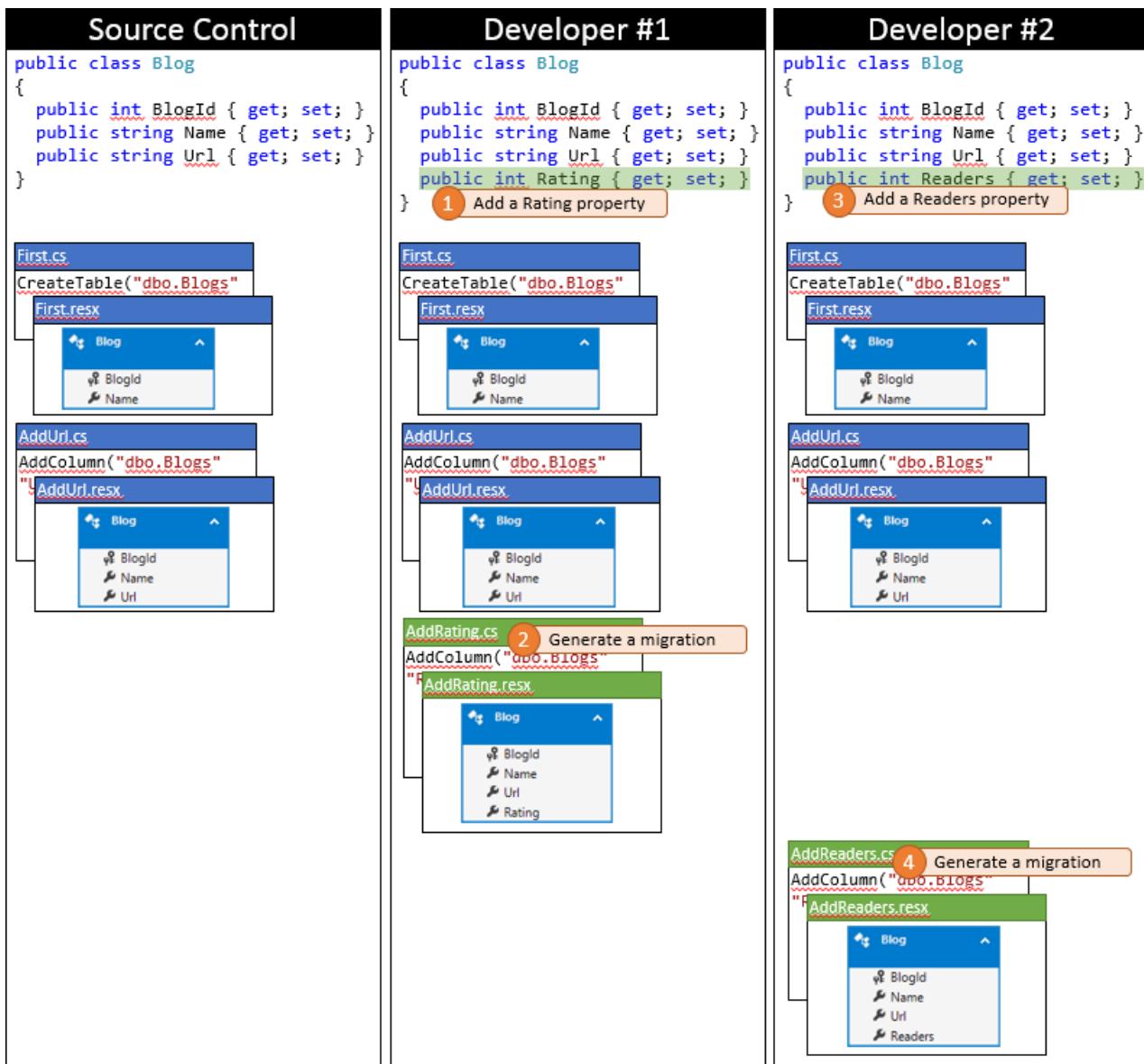
我们将通过多个更改跟踪 EF 模型和迁移。对于起始点，这两个开发人员已同步到源代码管理存储库，如下图所示。



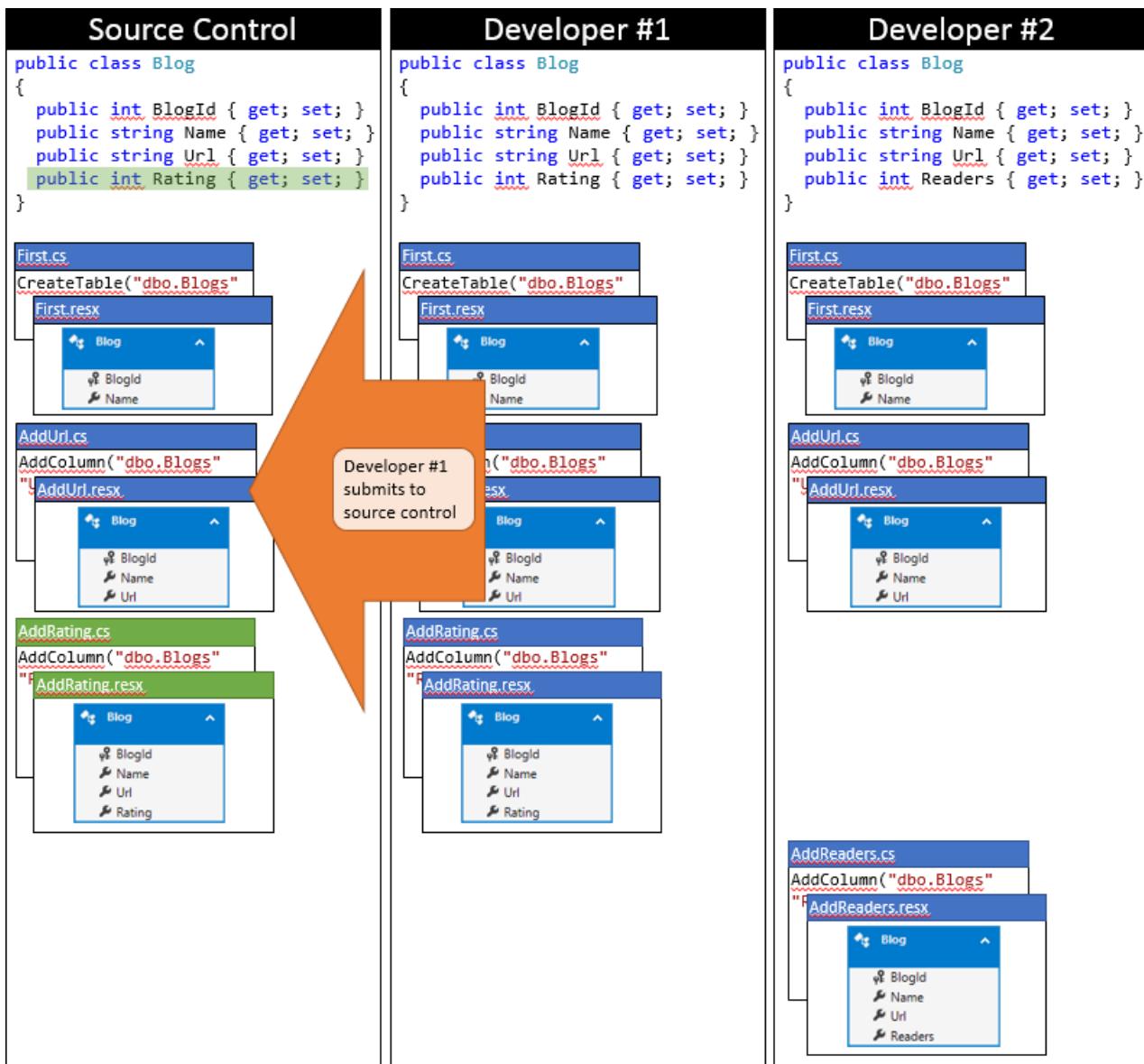
开发人员 #1 和开发人员 #2 现在会在其本地代码库中对 EF 模型进行一些更改。开发人员 #1 将分级属性添加到博客 – 并生成 AddRating 迁移，以将更改应用到数据库。开发人员 #2 将读者属性添加到博客 – 并生成相应的 AddReaders 迁移。这两个开发人员都运行更新数据库，以将更改应用到本地数据库，然后继续开发应用程序。

NOTE

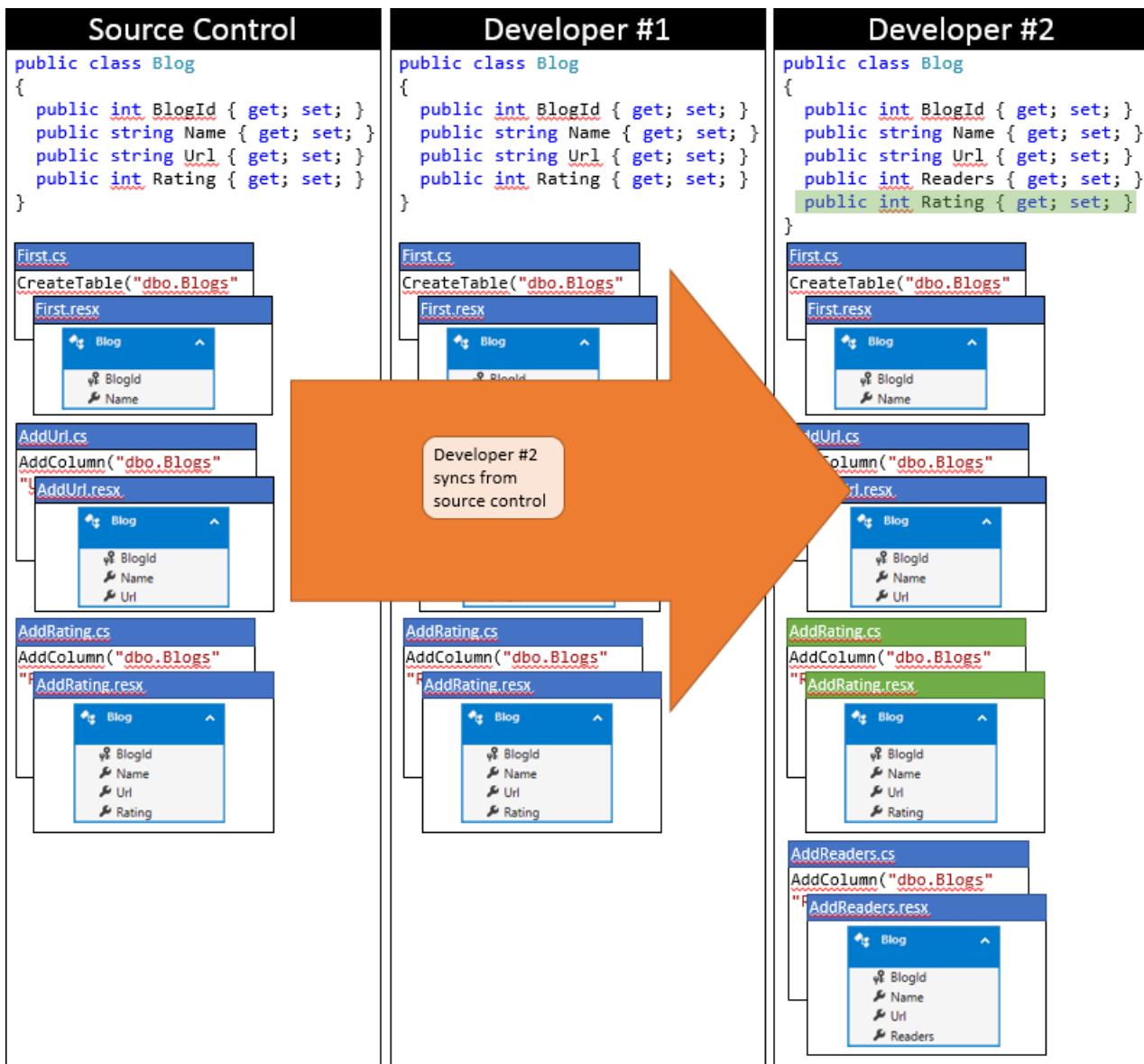
迁移以时间戳为前缀，因此我们的图形表示从开发人员 #2 开始迁移的 AddReaders 从开发人员 #1 开始迁移。开发人员 #1 或 #2 生成迁移首先对团队中工作的问题没有任何区别，或合并这些问题的过程，我们将在下一节中介绍。



这是一个幸运的日子，开发人员 #1，因为他们要首先提交更改。由于其他人在同步其存储库后未签入，因此他们只需提交其更改而无需执行任何合并。



现在，开发人员 #2 进行提交。它们不太幸运。由于其他人已在同步后提交了更改，因此他们将需要下拉更改并进行合并。源代码管理系统可能会自动将更改合并到代码级别，因为它们非常简单。在同步后，开发人员 #2 的本地存储库的状态如下图所示。



在此阶段，开发人员 #2 可以运行更新数据库，它将检测新的AddRating迁移（尚未应用于开发人员 #2 的数据库）并应用该迁移。现在，“评级”列将添加到“博客”表中，并且数据库与模型同步。

但有几个问题：

- 尽管“更新-数据库”将应用AddRating迁移，但它还会引发警告：无法更新数据库以匹配当前模型，因为存在挂起的更改，并且禁用了自动迁移... 问题在于，上一次迁移（AddReader）中存储的模型快照缺少博客上的“分级”属性（因为它不是生成迁移时模型的一部分）。Code First 检测到上一次迁移中的模型与当前模型不匹配，并引发警告。
- 运行应用程序会导致 InvalidOperationException，指出“bloggingcontext”上下文的模型在创建数据库后已发生更改。请考虑使用 Code First 迁移更新数据库... 同样，问题在于上一次迁移中存储的模型快照与当前模型不匹配。
- 最后，我们希望运行添加迁移现在会生成一个空迁移（因为没有要应用于数据库的更改）。但是，因为迁移会将当前模型与上一次迁移（缺少分级属性）的模型进行比较，所以它将真正基架另一次AddColumn调用以添加到分级列中。当然，此迁移将在更新数据库过程中失败，因为“分级”列已经存在。

解决合并冲突

好消息是，如果你对迁移的工作方式有所了解，就不难手动处理合并了。如果您跳过此部分，抱歉，您需要返回本文的剩余部分！

有两个选项，最简单的方法是生成一个将正确的当前模型作为快照的空白迁移。第二个选项是更新上一次迁移中的快照，使其具有正确的模型快照。第二个选项更难，不能用于每个方案中，但它也更清晰，因为它不涉及添加額

外的迁移。

选项1: 添加空白 "合并" 迁移

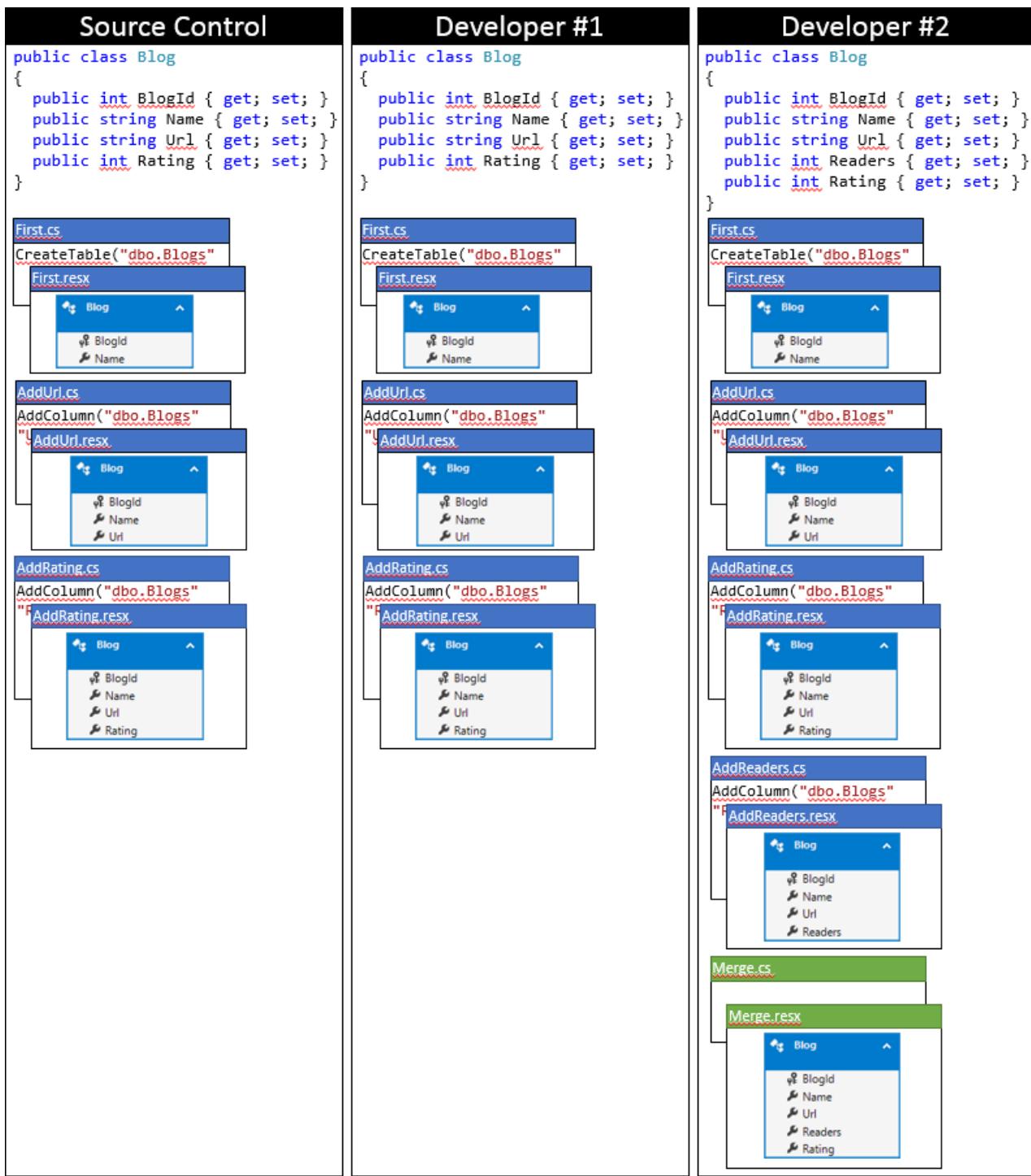
在此选项中, 我们将仅生成一个空白迁移, 目的是确保最新迁移中存储了正确的模型快照。

无论上次迁移的用户是谁, 都可以使用此选项。在本示例中, 我们已遵循开发人员 #2 正在处理合并, 并发生了过去的迁移。但是, 如果开发人员 #1 生成上次迁移, 则可以使用这些相同的步骤。如果涉及多个迁移, 则这些步骤也适用—我们只需查看两个步骤, 使其保持简单。

以下过程可用于此方法, 从你认识到需要从源控件同步的更改开始。

1. 确保已将本地代码库中的任何挂起的模型更改写入迁移。此步骤可确保在生成空白迁移时不会遗漏任何合法更改。
2. 与源代码管理同步。
3. 运行 "更新-数据库" 以应用其他开发人员已签入的任何新迁移。**注意:** 如果你没有从更新-数据库命令收到任何警告, 则没有来自其他开发人员的新迁移, 无需执行任何进一步的合并。
4. 运行添加迁移 <选择_名称> – IgnoreChanges (例如, 添加迁移合并– IgnoreChanges)。这会生成包含所有元数据(包括当前模型的快照)的迁移, 但在将当前模型与上一次迁移中的快照进行比较时, 会忽略它检测到的任何更改(这意味着你会获得一个空的向上和向下方法)。
5. 继续开发或提交到源代码管理(当然在运行单元测试后)。

下面是使用此方法后开发人员 #2 的本地代码库的状态。



选项2: 在上一次迁移中更新模型快照

此选项与选项1非常相似，但会删除额外的空白迁移，因为让我们面对自己的解决方案，需要额外的代码文件。

此方法仅在以下情况下可行：仅在本地代码库中存在最新的迁移，并且尚未提交到源控件（例如，如果执行合并的用户生成了最后一个迁移）。编辑其他开发人员可能已应用于其开发数据库（甚至更糟）的迁移的元数据可能会导致意外的副作用。在此过程中，我们将在本地数据库中回滚上一次迁移，并使用更新的元数据重新应用它。

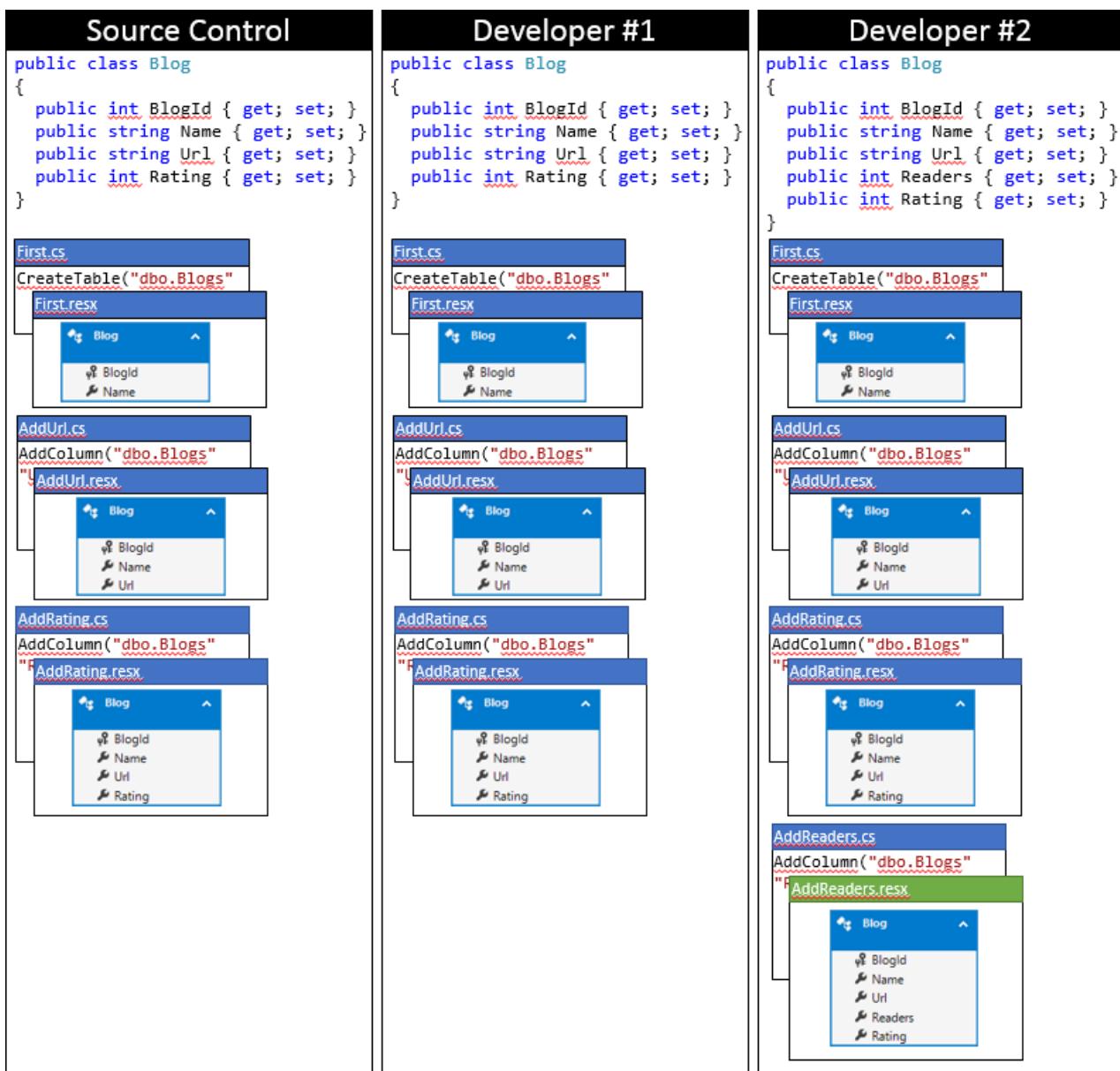
虽然最后的迁移需要位于本地代码库中，但在此过程中，迁移的数量或顺序并没有限制。可以从多个不同的开发人员进行多个迁移，相同的步骤同样适用—我们一直在寻找两个，使其保持简单。

以下过程可用于此方法，从你认识到需要从源控件同步的更改开始。

- 确保已将本地代码库中的任何挂起的模型更改写入迁移。此步骤可确保在生成空白迁移时不会遗漏任何合法更改。
- 与源代码管理同步。
- 运行“更新-数据库”以应用其他开发人员已签入的任何新迁移。**注意：**如果你没有从更新-数据库命令收到任何警告，则没有来自其他开发人员的新迁移，无需执行任何进一步的合并。

- 运行update-database – TargetMigration <second_上次_迁移> (在本示例中, 我们将是TargetMigration AddRating)。这会将数据库恢复到第二次迁移的状态, 这实际上是从数据库中 "取消应用" 最后一次迁移。**注意: 此步骤是为了安全地编辑迁移的元数据所必需的, 因为元数据也存储在数据库的 _MigrationsHistoryTable 中。这就是仅当最后一次迁移仅在本地代码库中时才应使用此选项的原因。如果其他数据库应用了上次迁移, 则还必须将其回滚并重新应用上次迁移以更新元数据。**
- 运行添加迁移 <完全_名称_包括_上次_迁移_的时间戳> (在本示例中, 我们的内容类似于添加迁移 201311062215252_AddReaders)。注意: 需要包含时间戳, 以便迁移知道要编辑现有迁移, 而不是新的基架。这将更新上一次迁移的元数据以匹配当前模型。当命令完成时, 你将收到以下警告, 但这正是你所希望的。"只有迁移" 201311062215252_AddReaders "的设计器代码已重新基架。若要重新基架整个迁移, 请使用-Force 参数。"
- 运行Update-Database, 以使用更新的元数据重新应用最新的迁移。
- 继续开发或提交到源代码管理(当然在运行单元测试后)。

下面是使用此方法后开发人员 #2 的本地代码库的状态。



摘要

在团队环境中使用 Code First 迁移时, 需要一些挑战。不过, 基本了解迁移的工作方式, 以及解决合并冲突的一些简单方法, 可以轻松地克服这些难题。

基本问题是最新迁移中存储的元数据不正确。这会导致 Code First 错误检测到当前模型和数据库架构不匹配, 也

不会在下一次迁移中基架错误代码。通过使用正确的模型生成空白迁移，或在最新的迁移中更新元数据，可以解决这种情况。

Model First

2020/3/12 •

此视频和分步演练提供使用实体框架 Model First 开发的简介。Model First 允许使用 Entity Framework Designer 创建新模型，然后从该模型生成数据库架构。模型存储在 EDMX 文件 (.edmx 扩展名) 中，可在 Entity Framework Designer 中查看和编辑。在应用程序中与之交互的类将自动从 EDMX 文件生成。

观看视频

此视频和分步演练提供使用实体框架 Model First 开发的简介。Model First 允许使用 Entity Framework Designer 创建新模型，然后从该模型生成数据库架构。模型存储在 EDMX 文件 (.edmx 扩展名) 中，可在 Entity Framework Designer 中查看和编辑。在应用程序中与之交互的类将自动从 EDMX 文件生成。

主讲人 :[Rowan Miller](#)

视频:[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

你将需要安装 Visual Studio 2010 或 Visual Studio 2012 才能完成此演练。

如果你使用的是 Visual Studio 2010，你还需要安装[NuGet](#)。

1. 创建应用程序

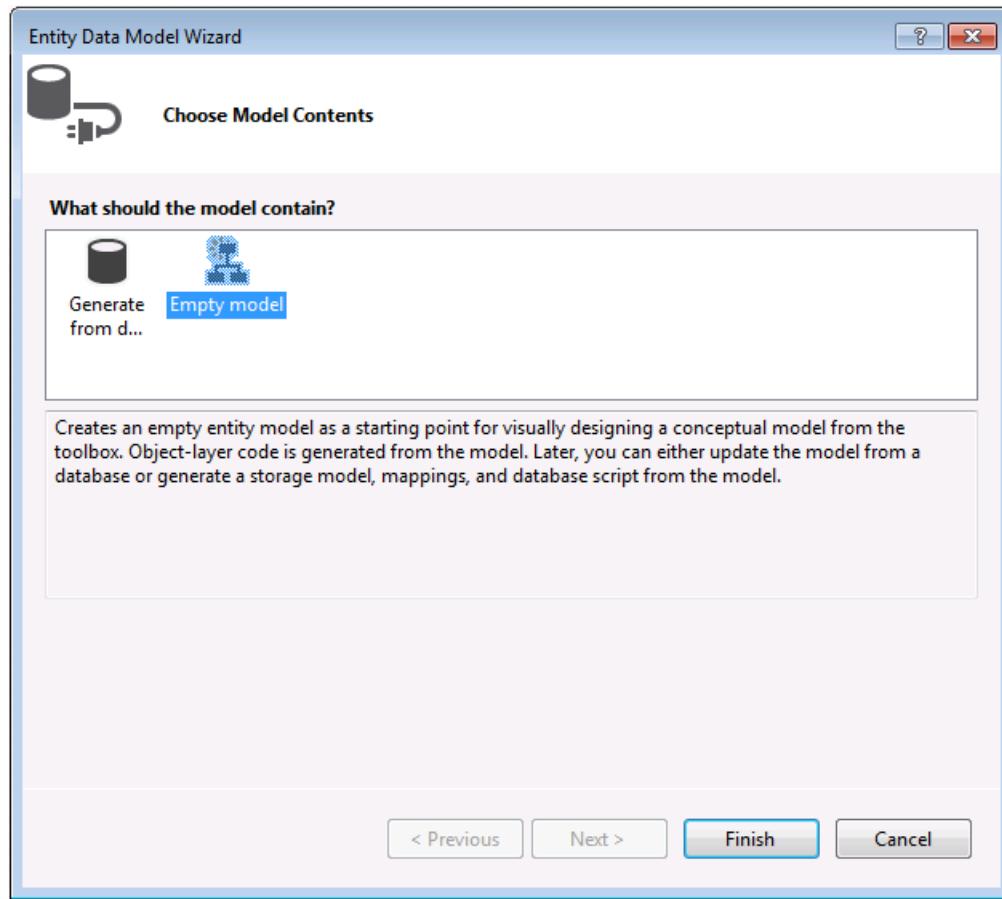
为了简单起见，我们将构建一个使用 Model First 执行数据访问的基本控制台应用程序：

- 打开 Visual Studio
- 文件->> 项目。
- 从左侧菜单和控制台应用程序选择Windows
- 输入**ModelFirstSample**作为名称
- 选择“确定”

2. 创建模型

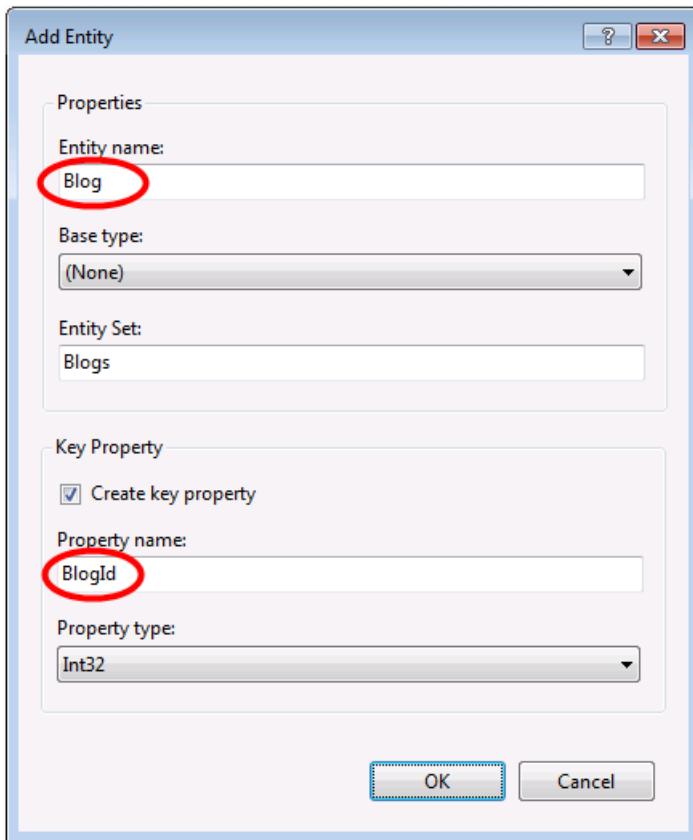
我们将使用在 Visual Studio 中包含的 Entity Framework Designer 来创建模型。

- 项目-> "添加新项 ..."
- 从左侧菜单中选择 "数据"，然后ADO.NET 实体数据模型
- 输入**BloggingModel**作为名称，然后单击 "确定"，这将启动实体数据模型向导
- 选择 "空模型"，然后单击 "完成"



将使用空白模型打开 Entity Framework Designer。现在，我们可以开始向模型添加实体、属性和关联。

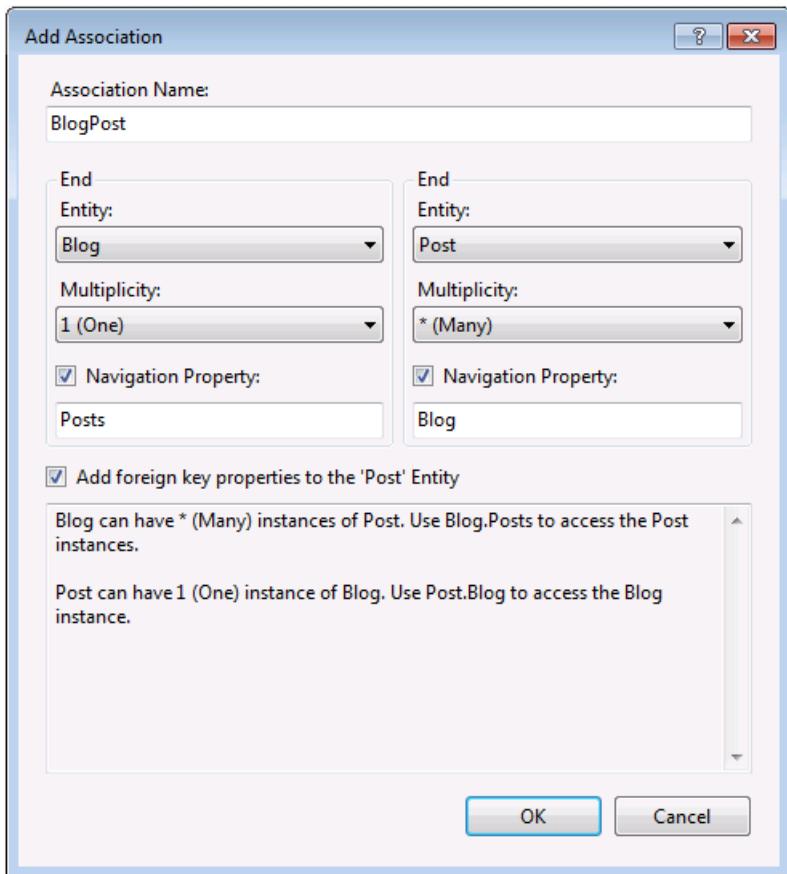
- 右键单击设计图面，然后选择 “属性”
- 在属性窗口将实体容器名称更改为 “bloggingcontext” 这是将为你生成的派生上下文的名称，上下文表示与数据库的会话，从而使我们能够查询和保存数据
- 右键单击设计图面，然后选择 “添加新的-> 实体 ...”
- 以 “实体名称” 和 “BlogId” 作为键名称输入博客，然后单击 “确定”



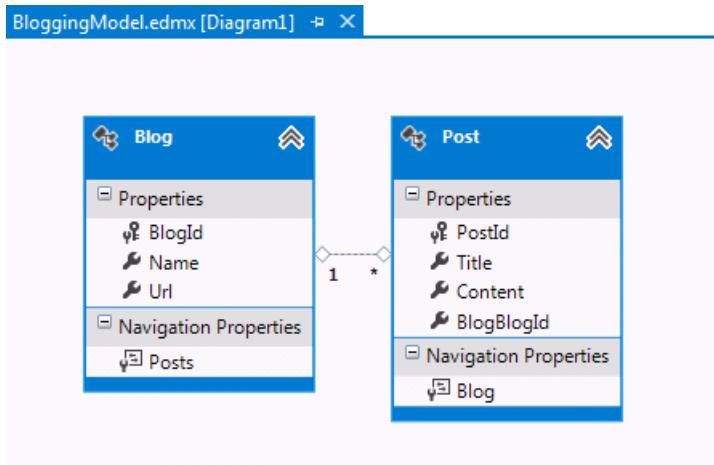
- 右键单击设计图面上的新实体，然后选择“添加新的> 标量属性”，输入name作为属性的名称。
- 重复此过程以添加Url属性。
- 右键单击设计图面上的“Url”属性，然后选择“属性”，在“属性窗口将可为 Null”的设置更改为True 这样就可以将博客保存到数据库，而无需向其分配Url
- 使用刚才了解到的技术，添加具有PostId键属性的Post实体
- 向Post实体添加Title和Content标量属性

现在我们有了几个实体，可以在它们之间添加关联(或关系)。

- 右键单击设计图面，然后选择“添加新> 关联 ... ”
- 将关系点的一端作为博客，使其重数为1，另一个终点使用 多个 的重数，这意味着，博客有多篇文章，张贴内容属于一个博客
- 确保选中“将外键属性添加到‘Post’实体”框，然后单击“确定”



现在，我们有了一个简单的模型，我们可以从它生成数据库并使用它来读取和写入数据。



Visual Studio 2010 中的其他步骤

如果使用的是 Visual Studio 2010，升级到最新版本的实体框架需要遵循一些额外步骤。升级很重要，因为它使你能够访问改进的 API 图面，更易于使用，并提供最新的 bug 修复。

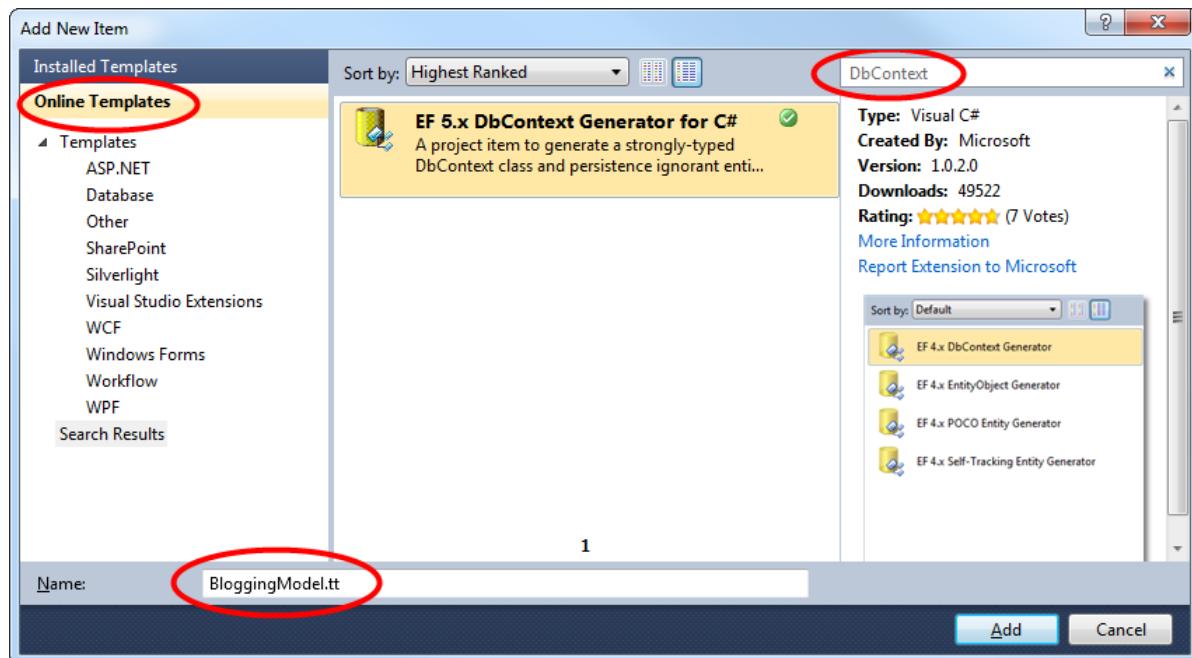
首先，我们需要从 NuGet 获取最新版本的实体框架。

- 项目-> 管理 NuGet 程序包 ... 如果你没有 “**管理 nuget 包 ...” 选项，则应安装[最新版本的 nuget *](#)
- 选择 "联机" 选项卡
- 选择EntityFramework包
- 单击“安装”

接下来，我们需要交换模型，以生成使用 DbContext API 的代码，这些代码是在实体框架的更高版本中引入的。

- 在 EF 设计器中右键单击模型的空位置，然后选择 "添加代码生成项 ... "
- 从左侧菜单中选择 "联机模板"，然后搜索DbContext

- 选择用于 C# 的 EF DbContext 生成器，输入BloggingModel作为名称，然后单击“添加”



3. 正在生成数据库

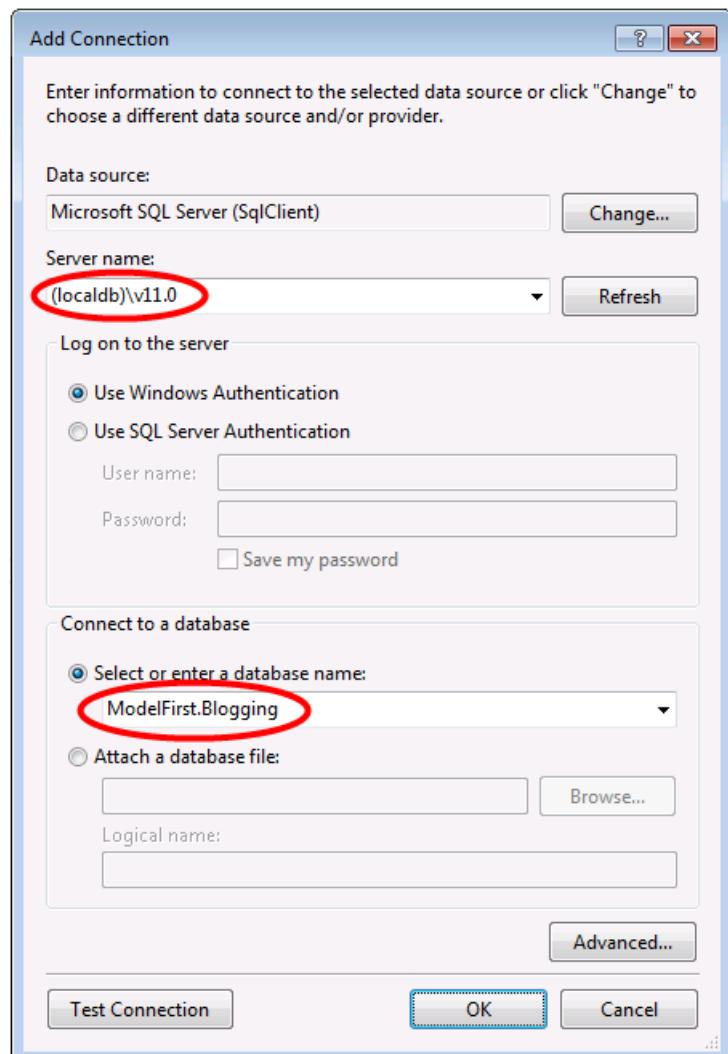
考虑到我们的模型，实体框架可以计算一个数据库架构，该架构将允许我们使用模型存储和检索数据。

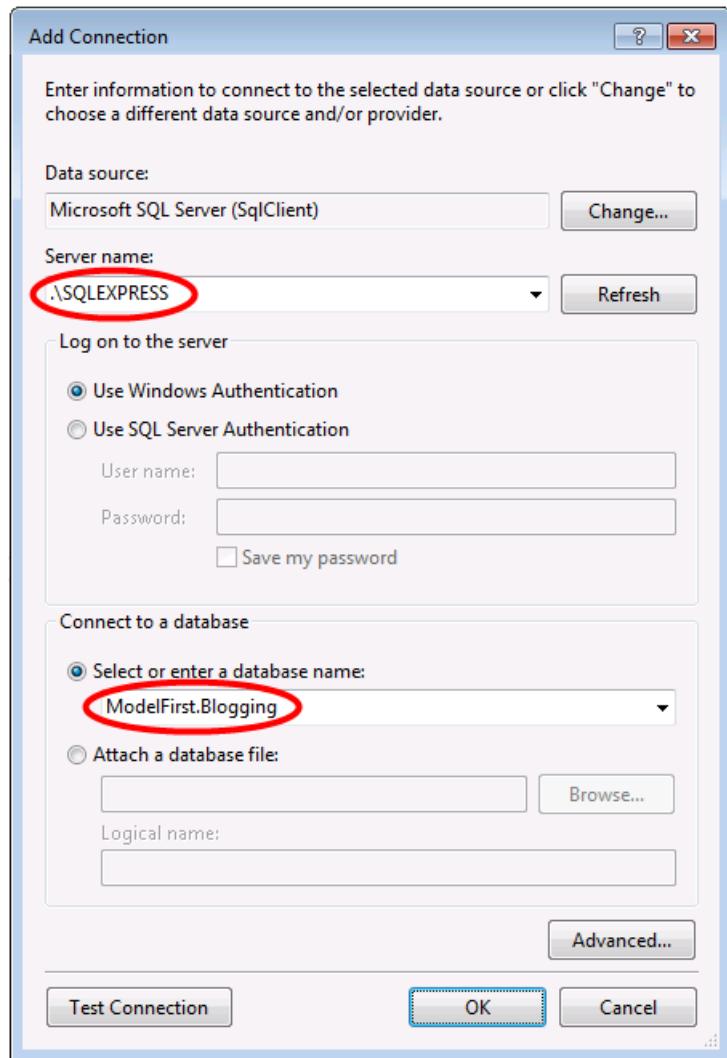
随 Visual Studio 一起安装的数据库服务器因安装的 Visual Studio 版本而异：

- 如果使用的是 Visual Studio 2010，则将创建 SQL Express 数据库。
- 如果使用的是 Visual Studio 2012，则将创建一个 LocalDB 数据库。

接下来，生成数据库。

- 右键单击设计图面，然后选择 “从模型生成数据库 ...”
- 单击 “新建连接 ...” 并指定 LocalDB 或 SQL Express，具体取决于所使用的 Visual Studio 的版本，请输入 ModelFirst 作为数据库名称。



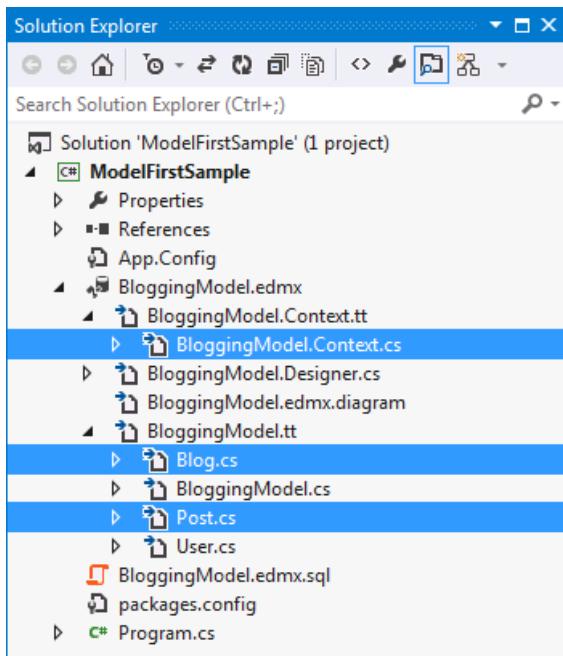


- 选择 "确定"，系统会询问您是否要创建新数据库，请选择 "是"
- 选择 "下一步"，Entity Framework Designer 将计算用于创建数据库架构的脚本
- 显示脚本后，单击 "完成"，脚本将添加到项目中并打开
- 右键单击该脚本，然后选择 "执行"，系统将提示您指定要连接到的数据库，指定 LocalDB 或 SQL Server Express，具体取决于您使用的 Visual Studio 的版本

4. 读取 & 写入数据

现在，我们有了一个模型，可以使用它来访问某些数据了。要用于访问数据的类将根据 EDMX 文件自动生成。

此屏幕快照来自 Visual Studio 2012，如果你使用的是 Visual Studio 2010，则 BloggingModel.tt 和 BloggingModel.Context.tt 文件将直接位于你的项目下，而不是在 EDMX 文件下嵌套。



在 Program.cs 中实现 Main 方法，如下所示。此代码创建一个新的上下文实例，然后使用它来插入新的博客。然后，它使用 LINQ 查询从按标题字母顺序排序的数据库中检索所有博客。

```
class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}
```

你现在可以运行该应用程序并对其进行测试。

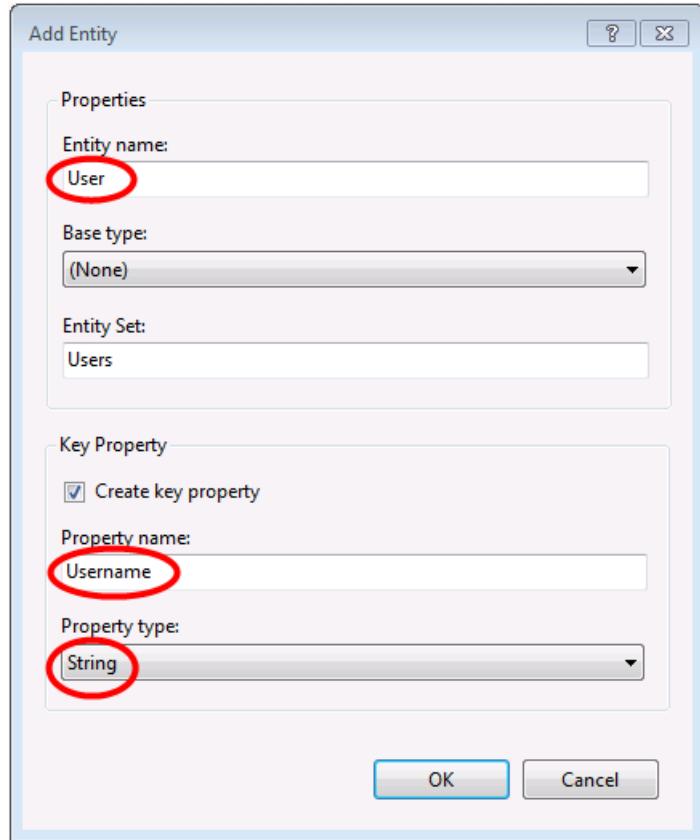
```
Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...
```

5. 处理模型更改

现在，可以对模型进行一些更改，当我们进行这些更改时，我们还需要更新数据库架构。

首先向模型添加新的用户实体。

- 添加新的用户实体名称，其中用户名为密钥名称，字符串用作密钥的属性类型



- 右键单击设计图面上的“用户名”属性，然后选择“属性”，在“属性窗口将”MaxLength“设置更改为50 这会将用户名中存储的数据限制为50个字符
- 向用户实体添加DisplayName标量属性

现在，我们有了一个更新的模型，我们可以更新数据库以适应新的用户实体类型。

- 右键单击设计图面，然后选择“从模型生成数据库 ...”，实体框架将计算一个脚本以根据更新后的模型重新创建架构。
- 单击“完成”
- 你可能会收到有关覆盖模型的现有 DDL 脚本以及映射和存储部分的警告，请单击“是”以显示这两个警告
- 将为您打开用于创建数据库的更新的 SQL 脚本
生成的脚本将删除所有现有的表，然后从头开始重新创建该架构。这可能适用于本地开发，但并不适合将更改推送到已部署的数据库。如果需要将更改发布到已部署的数据库，则需要编辑该脚本或使用架构比较工具来计算迁移脚本。
- 右键单击该脚本，然后选择“执行”，系统将提示您指定要连接到的数据库，指定 LocalDB 或 SQL Server Express，具体取决于您使用的 Visual Studio 的版本

Summary

在本演练中，我们介绍 Model First 开发，这允许我们在 EF 设计器中创建模型，然后从该模型生成数据库。然后，使用该模型从数据库中读取和写入一些数据。最后，我们更新了模型，然后重新创建了数据库架构以匹配模型。

Database First

2020/3/12 •

此视频和分步演练提供使用实体框架 Database First 开发的简介。Database First 允许从现有数据库对模型进行反向工程。模型存储在 EDMX 文件 (.edmx 扩展名) 中，可在 Entity Framework Designer 中查看和编辑。在应用程序中与之交互的类将自动从 EDMX 文件生成。

观看视频

此视频介绍了使用实体框架进行 Database First 开发的简介。Database First 允许从现有数据库对模型进行反向工程。模型存储在 EDMX 文件 (.edmx 扩展名) 中，可在 Entity Framework Designer 中查看和编辑。在应用程序中与之交互的类将自动从 EDMX 文件生成。

主讲人 : [Rowan Miller](#)

视频 : WMV | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

需要至少安装 Visual Studio 2010 或 Visual Studio 2012 才能完成此演练。

如果你使用的是 Visual Studio 2010，你还需要安装[NuGet](#)。

1. 创建现有数据库

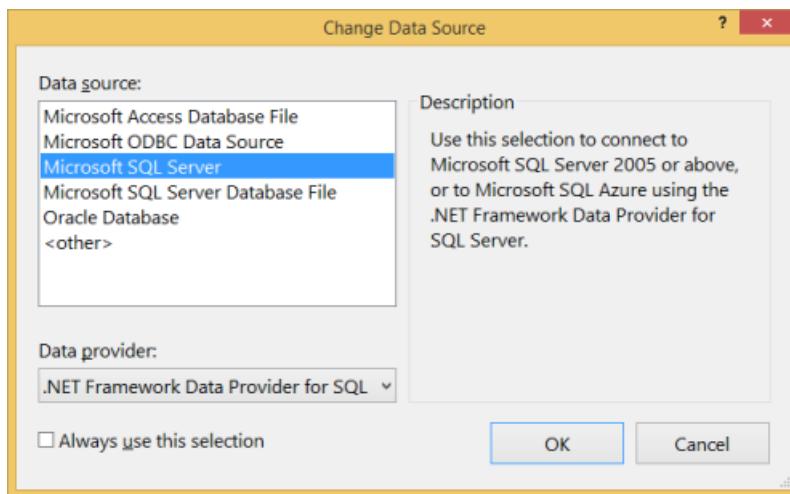
通常，当目标为现有数据库时，它将被创建，但在本演练中，我们需要创建一个要访问的数据库。

随 Visual Studio 一起安装的数据库服务器因安装的 Visual Studio 版本而异：

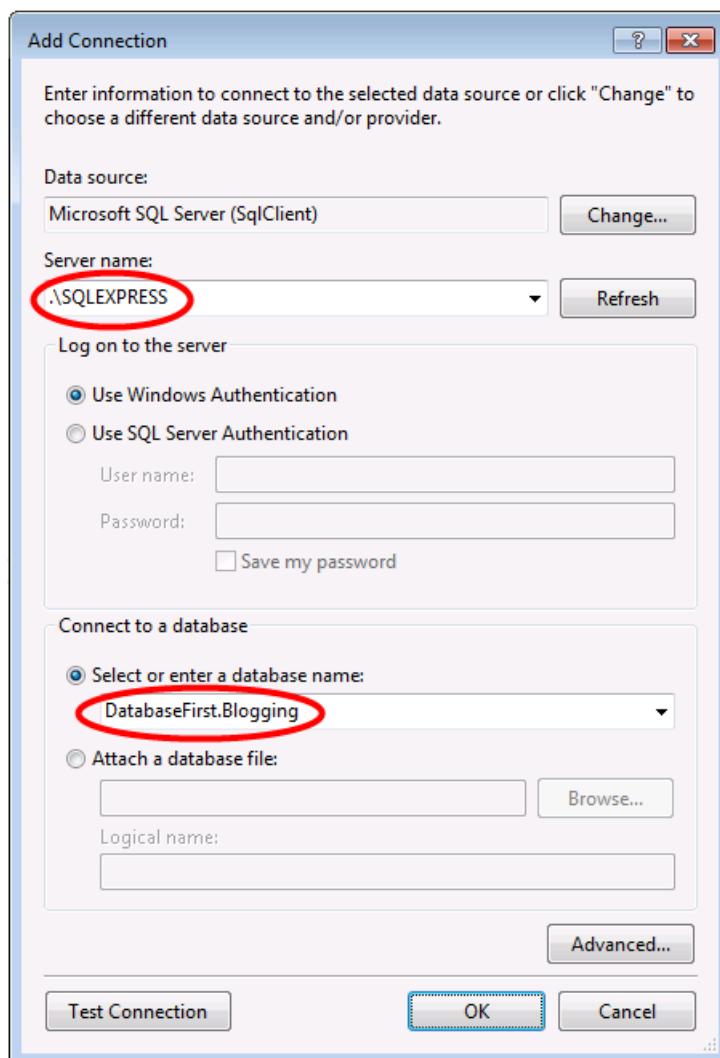
- 如果使用的是 Visual Studio 2010，则将创建 SQL Express 数据库。
- 如果使用的是 Visual Studio 2012，则将创建一个[LocalDB](#)数据库。

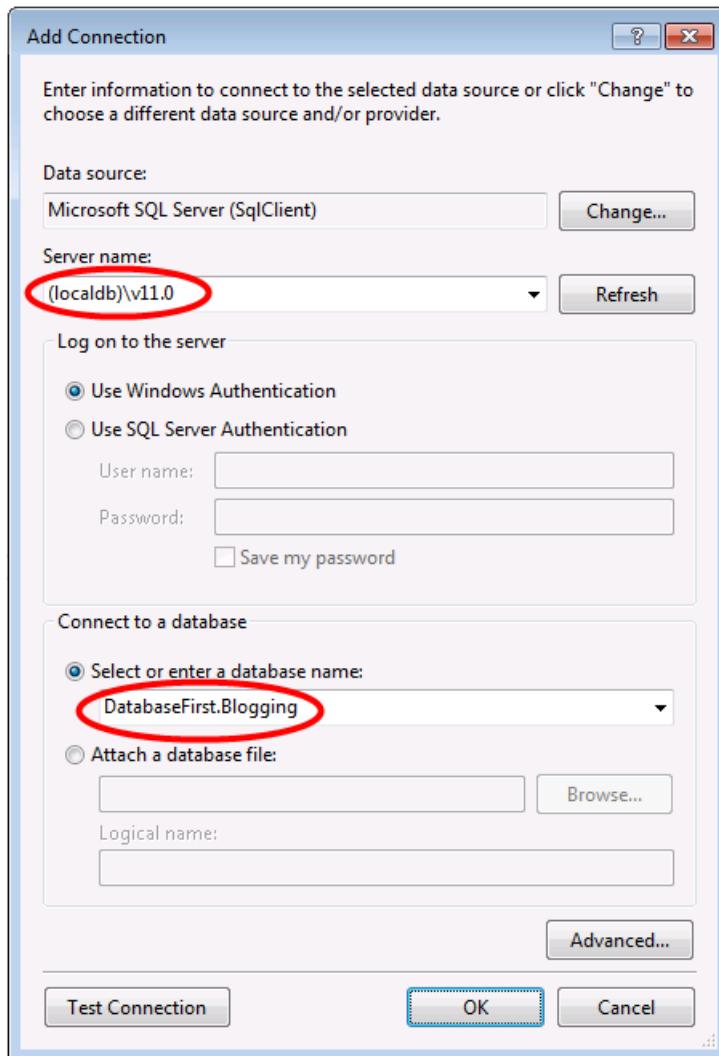
接下来，生成数据库。

- 打开 Visual Studio
- 视图-> 服务器资源管理器
- 右键单击 "数据连接-> 添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择 Microsoft SQL Server 作为数据源



- 连接到 LocalDB 或 SQL Express, 具体取决于已安装的数据, 并输入DatabaseFirst作为数据库名称





- 选择“确定”，系统会询问您是否要创建新数据库，请选择“是”



- 新数据库现在将出现在服务器资源管理器中，右键单击该数据库并选择“新建查询”
- 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择“执行”

```
CREATE TABLE [dbo].[Blogs] (
    [BlogId] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (200) NULL,
    [Url] NVARCHAR (200) NULL,
    CONSTRAINT [PK_dbo.Blogs] PRIMARY KEY CLUSTERED ([BlogId] ASC)
);

CREATE TABLE [dbo].[Posts] (
    [PostId] INT IDENTITY (1, 1) NOT NULL,
    [Title] NVARCHAR (200) NULL,
    [Content] NTEXT NULL,
    [BlogId] INT NOT NULL,
    CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
    CONSTRAINT [FK_dbo.Posts_dbo.Blogs_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [dbo].[Blogs] ([BlogId]) ON
DELETE CASCADE
);
```

2. 创建应用程序

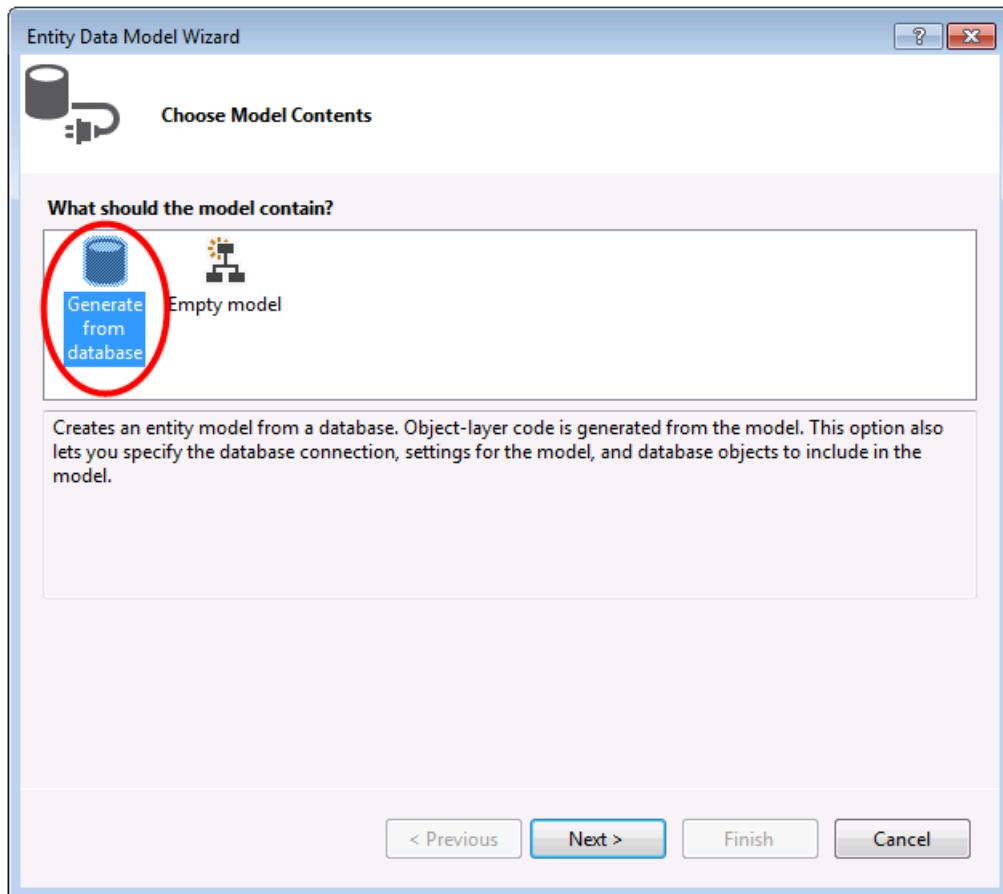
为了简单起见，我们将构建一个使用 Database First 执行数据访问的基本控制台应用程序：

- 打开 Visual Studio
- 文件->> 项目。
- 从左侧菜单和控制台应用程序选择Windows
- 输入DatabaseFirstSample作为名称
- 选择“确定”

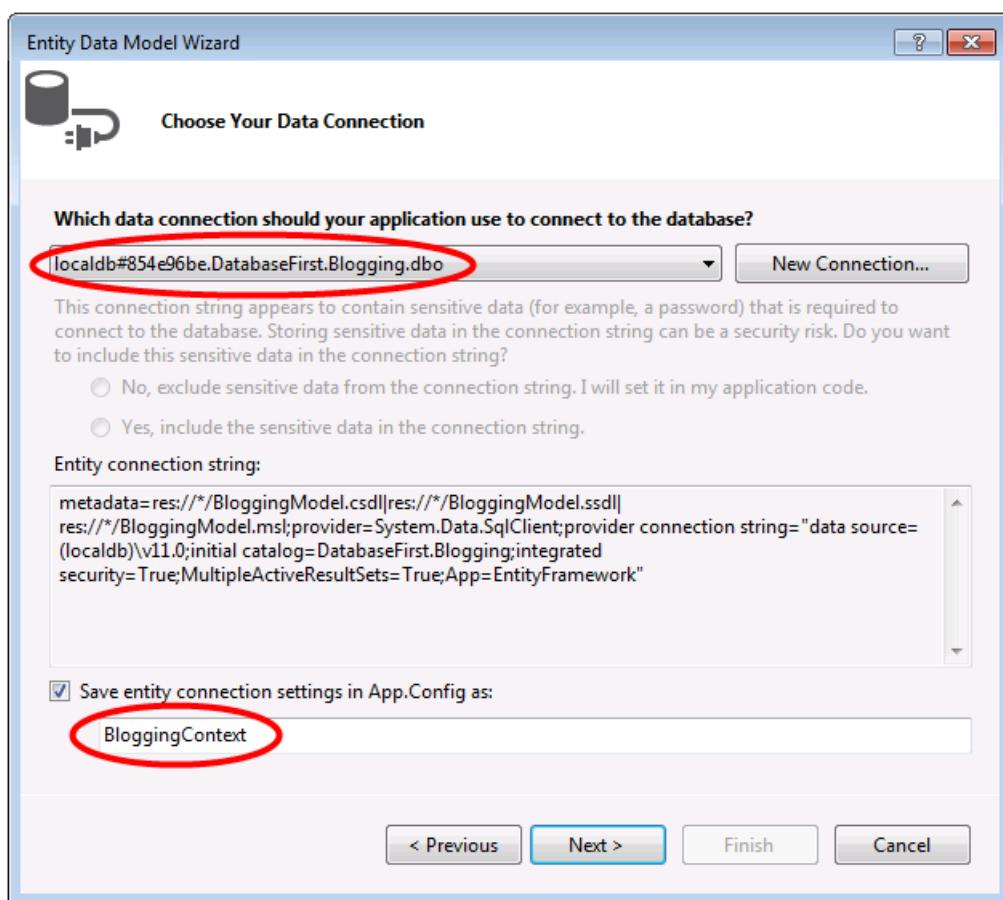
3. 反向工程模型

我们将使用在 Visual Studio 中包含的 Entity Framework Designer 来创建模型。

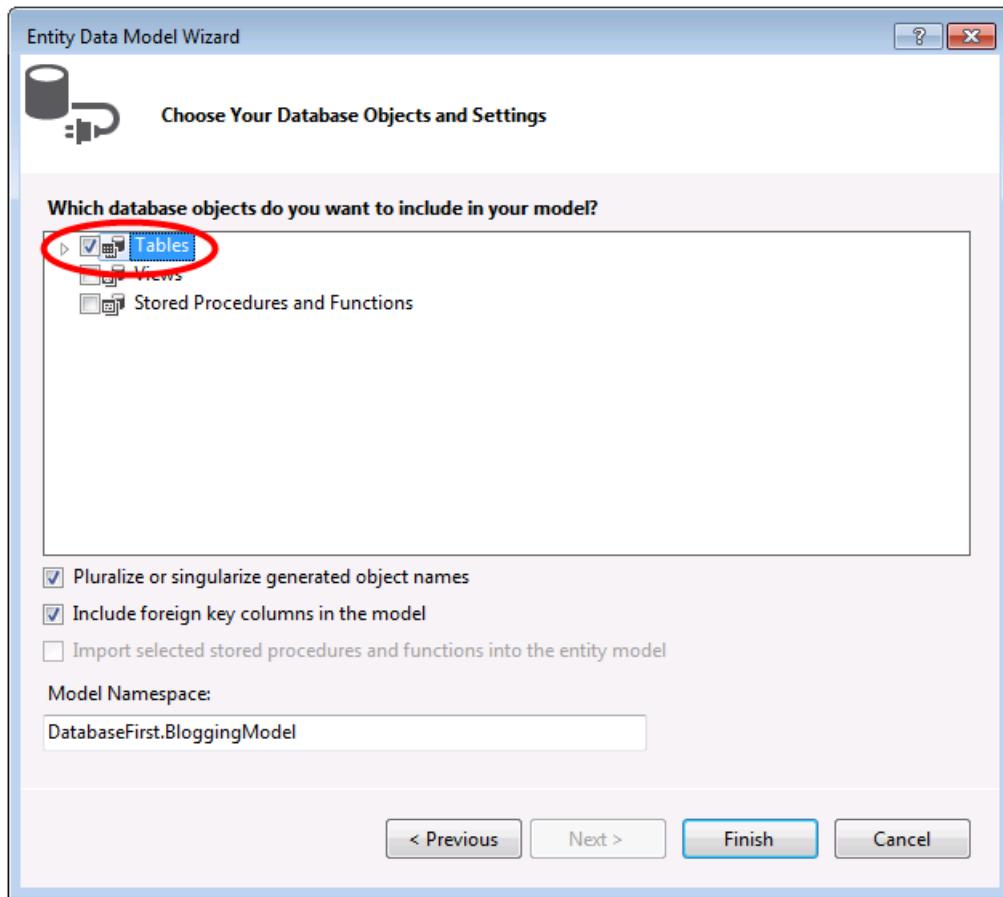
- 项目-> "添加新项 ..."
- 从左侧菜单中选择 "数据"，然后ADO.NET 实体数据模型
- 输入BloggingModel作为名称，然后单击 "确定"
- 这将启动实体数据模型向导
- 选择 "从数据库生成"，然后单击 "下一步"



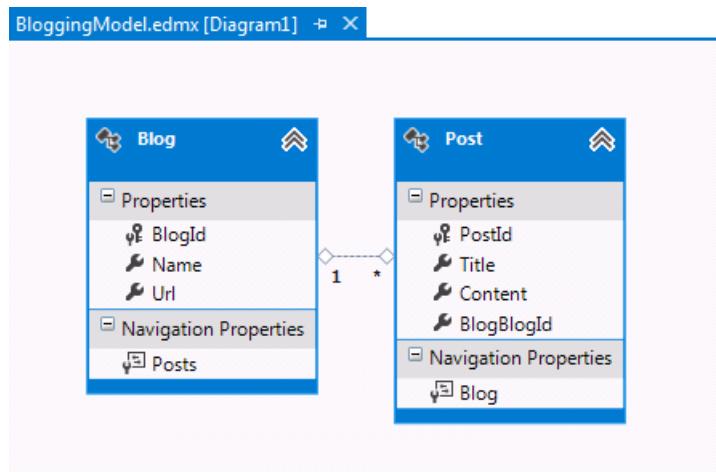
- 选择在第一部分中创建的数据库的连接，输入 " " bloggingcontext " " 作为连接字符串的名称，然后单击 "下一步"



- 单击 "表" 旁边的复选框以导入所有表，然后单击 "完成"



反向工程过程完成后，会将新模型添加到项目中，并打开，以便在 Entity Framework Designer 中查看。App.config 文件也已添加到您的项目中，其中包含数据库的连接详细信息。



Visual Studio 2010 中的其他步骤

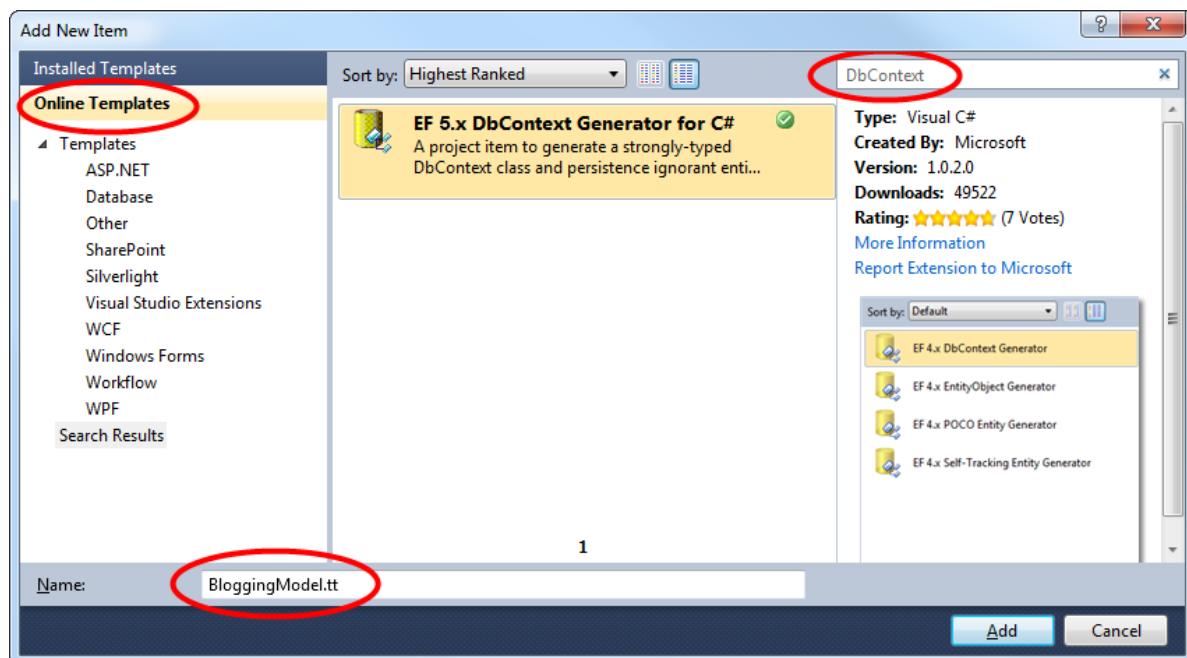
如果使用的是 Visual Studio 2010，升级到最新版本的实体框架需要遵循一些额外步骤。升级很重要，因为它使你能够访问改进的 API 图面，更易于使用，并提供最新的 bug 修复。

首先，我们需要从 NuGet 获取最新版本的实体框架。

- 项目-> 管理 NuGet 程序包 ... 如果你没有 “**管理 nuget 包 ...*” 选项，则应安装[最新版本的 nuget *](#)
- 选择“联机”选项卡
- 选择EntityFramework包
- 单击“安装”

接下来，我们需要交换模型，以生成使用 DbContext API 的代码，这些代码是在实体框架的更高版本中引入的。

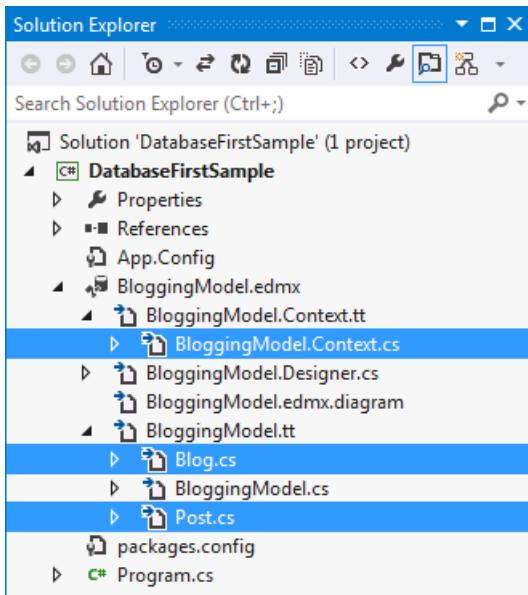
- 在 EF 设计器中右键单击模型的空位置，然后选择“添加代码生成项 ...”
- 从左侧菜单中选择“联机模板”，然后搜索DbContext
- 选择用于 C# 的 EF DbContext 生成器，输入BloggingModel作为名称，然后单击“添加”



4. 读取 & 写入数据

现在，我们有了一个模型，可以使用它来访问某些数据了。要用于访问数据的类将根据 EDMX 文件自动生成。

此屏幕快照来自 Visual Studio 2012，如果你使用的是 Visual Studio 2010，则 BloggingModel.tt 和 BloggingModel.Context.tt 文件将直接位于你的项目下，而不是在 EDMX 文件下嵌套。



在 Program.cs 中实现 Main 方法，如下所示。此代码创建一个新的上下文实例，然后使用它来插入新的博客。然后，它使用 LINQ 查询从按标题字母顺序排序的数据库中检索所有博客。

```

class Program
{
    static void Main(string[] args)
    {
        using (var db = new BloggingContext())
        {
            // Create and save a new Blog
            Console.Write("Enter a name for a new Blog: ");
            var name = Console.ReadLine();

            var blog = new Blog { Name = name };
            db.Blogs.Add(blog);
            db.SaveChanges();

            // Display all Blogs from the database
            var query = from b in db.Blogs
                        orderby b.Name
                        select b;

            Console.WriteLine("All blogs in the database:");
            foreach (var item in query)
            {
                Console.WriteLine(item.Name);
            }

            Console.WriteLine("Press any key to exit...");
            Console.ReadKey();
        }
    }
}

```

你现在可以运行该应用程序并对其进行测试。

```

Enter a name for a new Blog: ADO.NET Blog
All blogs in the database:
ADO.NET Blog
Press any key to exit...

```

5. 处理数据库更改

现在，需要对数据库架构进行一些更改。我们进行这些更改时，我们还需要更新模型以反映这些更改。

第一步是对数据库架构进行一些更改。我们要将用户表添加到该架构中。

- 在服务器资源管理器中右键单击**DatabaseFirst**数据库，然后选择“新建查询”
- 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择“执行”

```

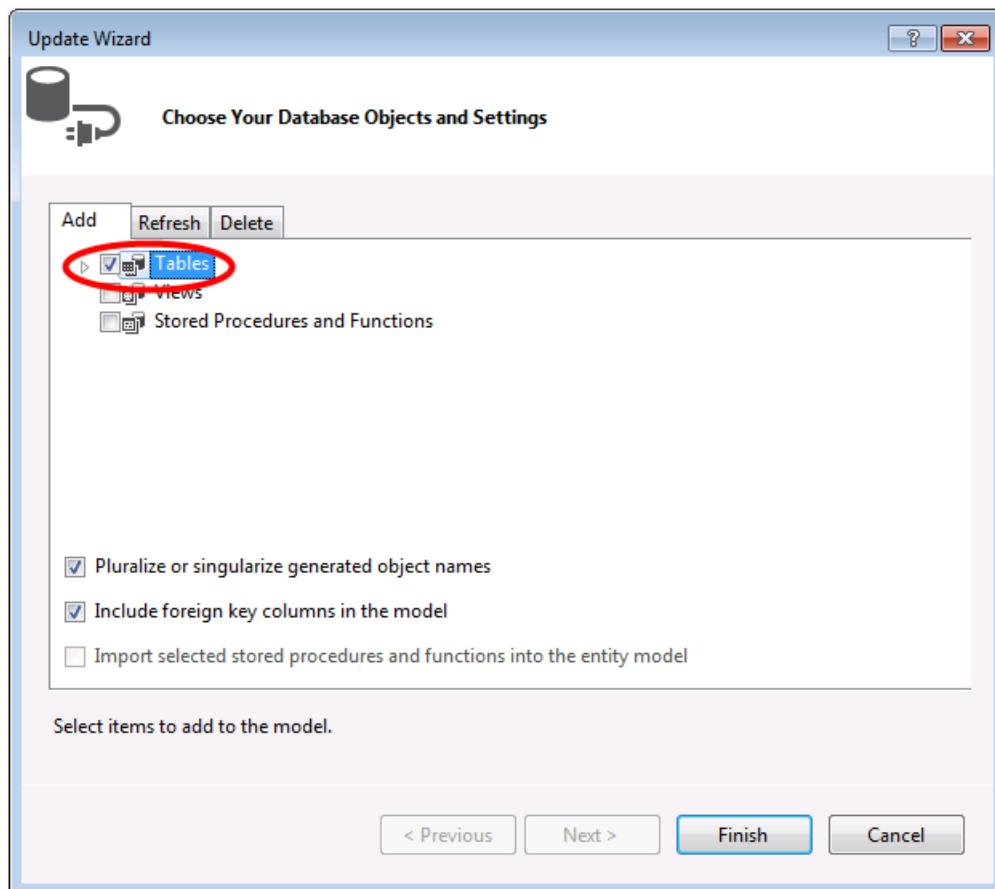
CREATE TABLE [dbo].[Users]
(
    [Username] NVARCHAR(50) NOT NULL PRIMARY KEY,
    [DisplayName] NVARCHAR(MAX) NULL
)

```

现在，架构已更新，现在可以用这些更改更新模型了。

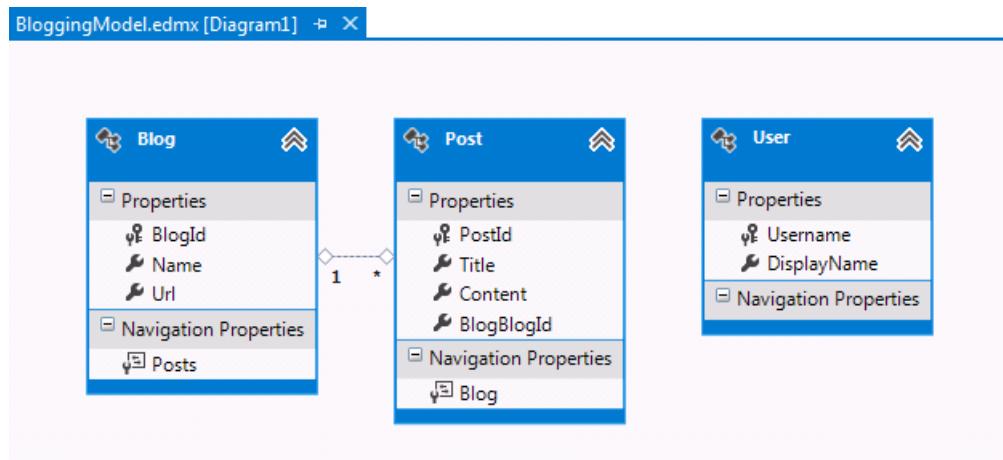
- 在 EF 设计器中右键单击模型的空白点，然后选择“从数据库更新模型...”，这将启动更新向导
- 在更新向导的“添加”选项卡上，选中“表”旁边的框，这表示要从架构中添加任何新表。“刷新”选项卡显示模型中的所有现有表，这些表将在更新过程中检查更改。“删除”选项卡显示已从架构中删除的所有表，并将作

为更新的一部分从模型中删除。这两个选项卡上的信息是自动检测的，仅提供提供信息，无法更改任何设置。



- 在更新向导中单击 "完成"

模型现在已更新为包含新的用户实体，该实体映射到添加到数据库中的用户表。



Summary

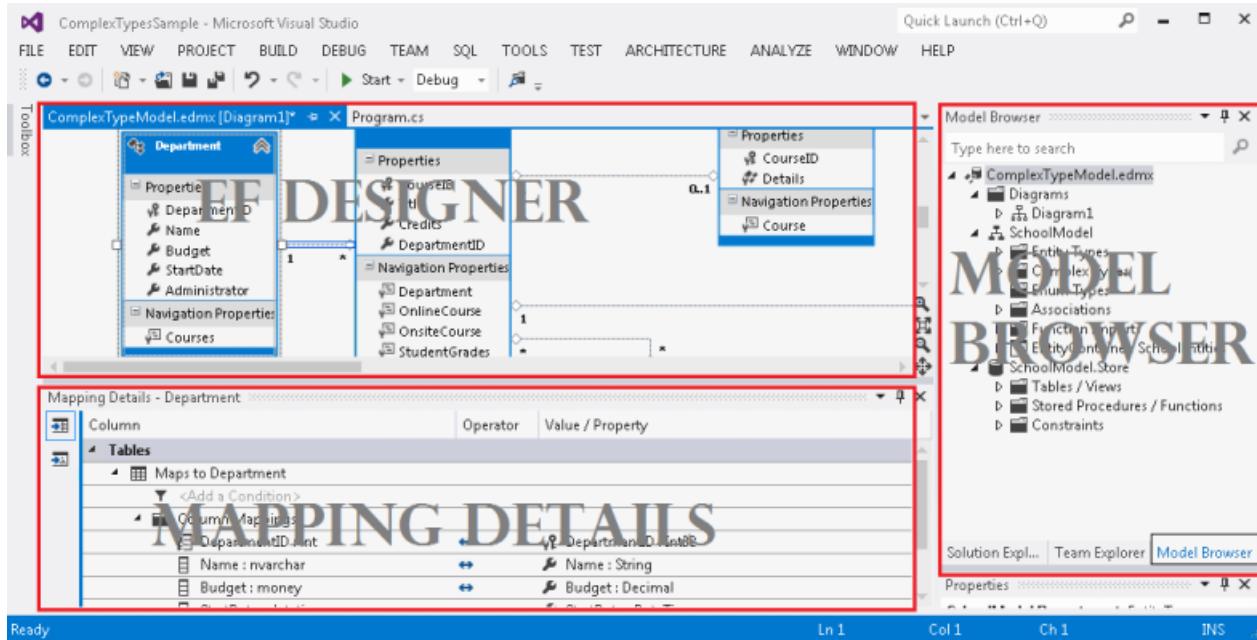
在本演练中，我们将讨论 Database First 开发，这允许我们基于现有数据库在 EF 设计器中创建模型。然后，使用该模型从数据库中读取和写入一些数据。最后，我们更新了模型以反映对数据库架构所做的更改。

复杂类型-EF 设计器

2020/3/11 ·

本主题演示如何将复杂类型映射到 Entity Framework Designer (EF 设计器) 以及如何查询包含复杂类型属性的实体。

下图显示了在使用 EF 设计器时使用的主窗口。



NOTE

在生成概念模型时，有关未映射的实体和关联的警告可能会显示在“错误列表”中。您可以忽略这些警告，因为在您选择从模型生成数据库后，错误将消失。

什么是复杂类型

复杂类型是实体类型的非标量属性，实体类型允许在实体内组织标量属性。与实体相似，复杂类型由标量属性或者其他复杂类型属性组成。

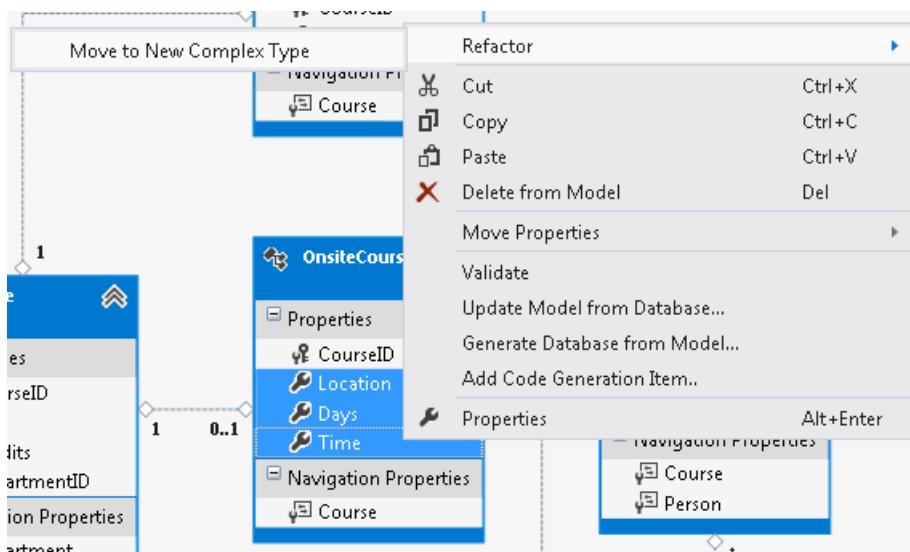
使用表示复杂类型的对象时，请注意以下事项：

- 复杂类型没有键，因此不能独立存在。复杂类型只能作为实体类型或其他复杂类型的属性而存在。
- 复杂类型不能参与关联且不能包含导航属性。
- 复杂类型属性不能为 null。当调用DbContext 并且遇到 null 复杂对象时，会发生 **InvalidOperationException**。复杂对象的标量属性可以为 null。
- 复杂类型不能从其他复杂类型继承。
- 必须将复杂类型定义为类。
- 调用DetectChanges时，EF 检测对复杂类型对象上的成员所做的更改。调用以下成员时，实体框架自动调用DetectChanges : DbSet、 DbSet、 DbSet、 DbSet、 DbSet、 DbContext、 DbContext、 GetValidationErrors、 DbContext、 DbChangeTracker、。

将实体的属性重构为新的复杂类型

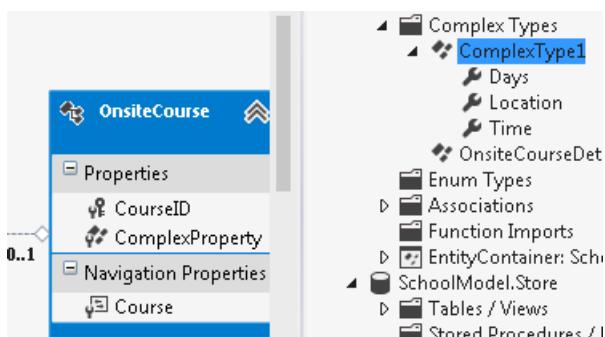
如果概念模型中已经有一个实体，可能想要将某些属性重构为复杂类型属性。

在设计器图面上，选择一个或多个实体属性（不包括导航属性），然后右键单击并选择“重构-> 移动到新的复杂类型”。



具有选定属性的新复杂类型将添加到模型浏览器。此复杂类型会被赋予一个默认名称。

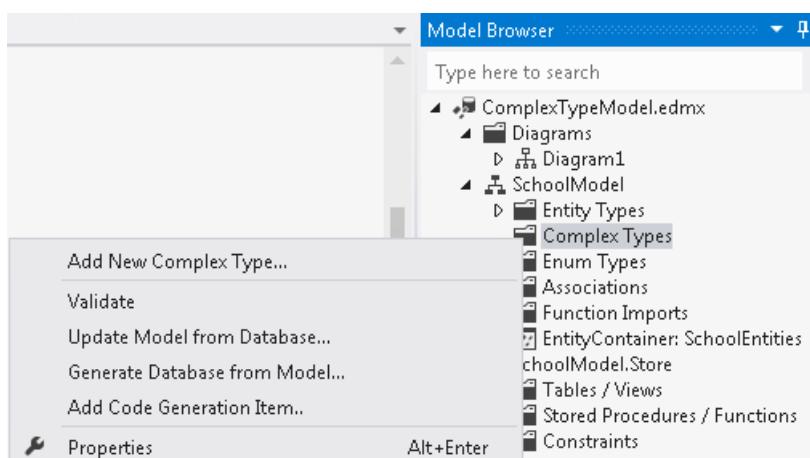
新创建类型的复杂属性将替换选定属性。所有属性映射都将保留。



创建新的复杂类型

你还可以创建不包含现有实体的属性的新复杂类型。

右键单击“模型浏览器”中的“复杂类型”文件夹，指向“Add New 复杂类型 ...”。或者，你可以选择“复杂类型”文件夹，并按键盘上的“插入”键。



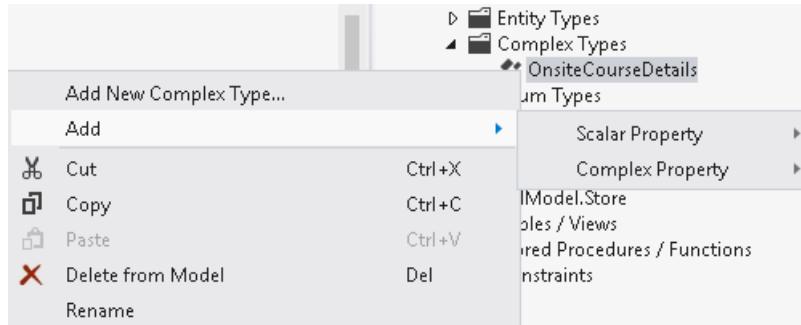
新复杂类型将添加到具有默认名称的文件夹。你现在可以将属性添加到该类型。

向复杂类型添加属性

复杂类型的属性可以是标量类型或现有的复杂类型。但是，复杂类型属性无法具有循环引用。例如，复杂类型 `OnsiteCourseDetails` 不能具有复杂类型 `OnsiteCourseDetails` 的属性。

可以通过下面列出的任何方式向复杂类型添加属性。

- 在模型浏览器中右键单击复杂类型，指向“添加”，再指向“标量属性”或“复杂属性”，然后选择所需的属性类型。或者，您可以选择一种复杂类型，然后按下键盘上的 **Insert** 键。



新属性将添加到具有默认名称的复杂类型。

- 或 -
- 右键单击 EF 设计器图面上的实体属性，选择“复制”，然后在“模型浏览器”中右键单击复杂类型，然后选择“粘贴”。

重命名复杂类型

重命名复杂类型时，将通过项目更新对类型的所有引用。

- 在模型浏览器中，慢慢地双击复杂类型。名称将选定并处于编辑模式。
- 或 -
- 在模型浏览器中右键单击复杂类型，然后选择“重命名”。
- 或 -
- 在模型浏览器中选择复杂类型，按 F2 键。
- 或 -
- 在模型浏览器中右键单击复杂类型，然后选择“属性”。在“属性”窗口中编辑该名称。

将现有复杂类型添加到实体，并将其属性映射到表列

- 右键单击某个实体，指向“添加新项”，然后选择“复杂属性”。具有默认名称的复杂类型属性将添加到该实体。默认类型(从现有复杂类型中选择)将指派给该属性。
- 将所需的类型分配给“属性”窗口中的属性。在将复杂类型属性添加到实体后，必须将其属性映射到表列。
- 右键单击设计图面或 模型浏览器 中的某个实体类型 然后选择“表映射”。表映射显示在“窗口中的”映射详细信息“窗口中。
- 展开“映射”，以 <表名称> “节点。此时将显示 节点的 列映射。
- 展开“列映射”节点。出现一个包含表中所有列的列表。列映射的默认属性(如果有)将在“值/属性”标题下列出。
- 选择要映射的列，然后右键单击相应的 值/属性 “字段。出现一个包含所有标量属性的下拉列表。

7. 选择适当的属性。

The screenshot shows the EntityDataSource configuration interface. On the left, there's a navigation pane with 'Properties' and 'Navigation Properties' sections. In the main area, there's a table mapping section titled 'Mapping Details - OnsiteCourse'. It lists columns from the 'Details' table being mapped to properties of the 'OnsiteCourse' entity. The columns listed are CourseID, Location, Days, and Time, each mapped to their respective properties in the 'OnsiteCourse' entity.

Column	Operator	Value / Property
CourseID : nvarchar	=	CourseID : Int32
Location : nvarchar	=	Details.Location : String
Days : nvarchar	=	Details.Days : String
Time : smalldatetime	=	Details.Time : DateTime

8. 对每一个表列重复步骤 6 和 7。

NOTE

若要删除列映射，请选择要映射的列，然后单击“”字段。然后，从下拉列表中选择“”。

将函数导入映射到复杂类型

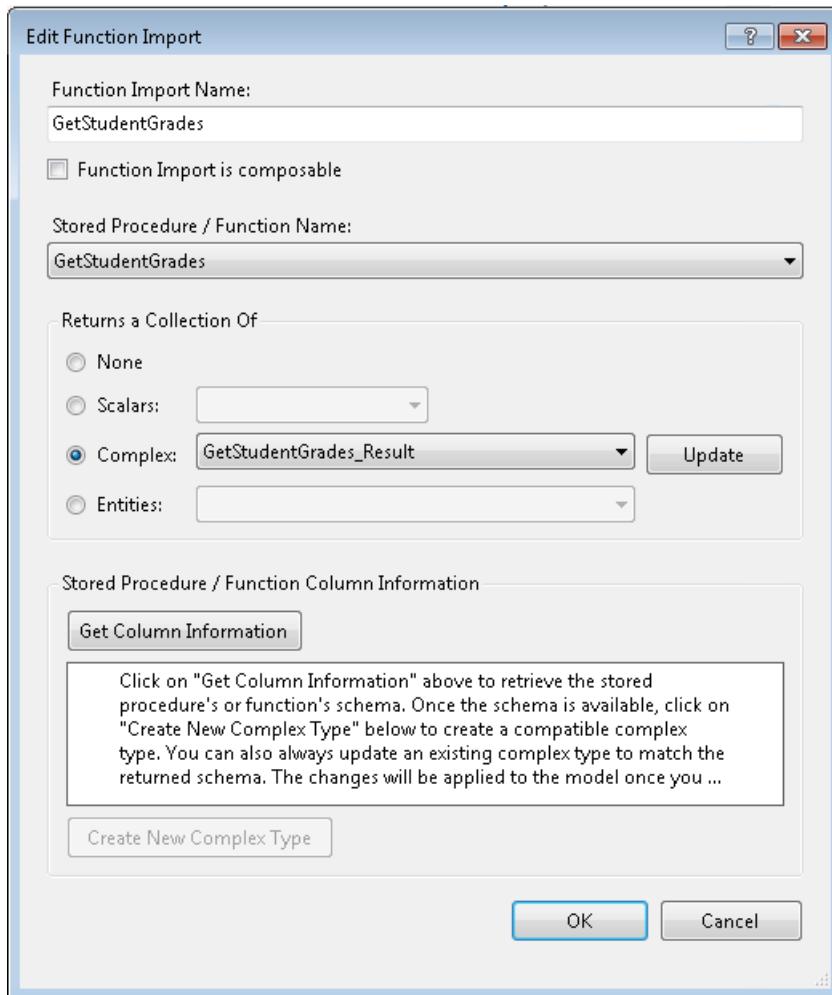
函数导入基于存储过程。若要将函数导入映射到复杂类型，相应存储过程返回的列必须在数量上与复杂类型的属性匹配并且必须具有与属性类型兼容的存储类型。

- 双击要映射到复杂类型的已导入函数。

The screenshot shows the Model Browser interface. The 'Function Imports' section is expanded, showing a list of functions. One function, 'GetStudentGrades', is highlighted with a blue selection bar. This indicates it is currently selected for configuration.

- 填入新函数导入的设置，如下所示：

- 在“存储过程名称”字段中指定要为其创建函数导入的存储过程字段。此字段是一个下拉列表，其中显示存储模型中的所有存储过程。
- 在“函数导入名称”字段中指定函数导入的名称。
- 选择“复杂”作为返回类型，然后通过从下拉列表中选择适当的类型来指定特定的复杂返回类型。



- 单击“确定”。此时将在概念模型中创建函数导入项。

自定义函数导入的列映射

- 右键单击模型浏览器中的函数导入，然后选择“函数导入映射”。此时将显示“映射详细信息”窗口，其中显示函数导入的默认映射。箭头指示列值和属性值之间的映射。默认情况，假设列名称与复杂类型的属性名称相同。默认的列名称以灰色文本显示。
- 如有必要，请更改列名称以匹配由对应于函数导入的存储过程返回的列名称。

删除复杂类型

删除复杂类型时，从概念模型删除类型，并且将删除该类型所有实例的映射。但是，不更新对类型的引用。例如，如果实体具有类型为 ComplexType1 的复杂类型属性，并且在模型浏览器中删除了 ComplexType1，则不会更新相应的实体属性。模型将不会验证，因为它包含引用已删除复杂类型的实体。可以使用实体设计器更新或删除对已删除复杂类型的引用。

- 在模型浏览器中右键单击复杂类型，然后选择“删除”。
- 或 -
- 在模型浏览器中选择复杂类型，然后按键盘上的“Delete”键。

查询包含复杂类型属性的实体

下面的代码演示如何执行一个查询，该查询返回包含复杂类型属性的实体类型对象的集合。

```
using (SchoolEntities context = new SchoolEntities())
{
    var courses =
        from c in context.OnsiteCourses
        order by c.Details.Time
        select c;

    foreach (var c in courses)
    {
        Console.WriteLine("Time: " + c.Details.Time);
        Console.WriteLine("Days: " + c.Details.Days);
        Console.WriteLine("Location: " + c.Details.Location);
    }
}
```

枚举支持-EF 设计器

2020/3/12 •

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

此视频和分步演练演示了如何将枚举类型与 Entity Framework Designer 一起使用。它还演示了如何在 LINQ 查询中使用枚举。

本演练将使用 Model First 创建新的数据库，但 EF 设计器也可以与[Database First](#)工作流一起用于映射到现有数据库。

实体框架5中引入了枚举支持。若要使用枚举、空间数据类型和表值函数等新功能，则必须以 .NET Framework 4.5 为目标。默认情况下，Visual Studio 2012 面向 .NET 4.5。

在实体框架中，枚举可以具有以下基础类型:[Byte](#)、[Int16](#)、[Int32](#)、[Int64](#)或[SByte](#)。

观看视频

此视频演示如何将枚举类型与 Entity Framework Designer 一起使用。它还演示了如何在 LINQ 查询中使用枚举。

主讲人 :Julia Kornich

视频 :WMV | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

你将需要安装 Visual Studio 2012、旗舰版、高级版、专业版或 Web Express edition 才能完成此演练。

设置项目

1. 打开 Visual Studio 2012
2. 在 "文件" 菜单上，指向 "新建"，然后单击 "项目"
3. 在左窗格中，单击 "Visual C#"，然后选择控制台模板
4. 输入EnumEFDesigner作为项目名称，然后单击 "确定"

使用 EF 设计器创建新模型

1. 右键单击 "解决方案资源管理器" 中的项目名称，指向"添加"，然后单击"新建项"
2. 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"
3. 输入EnumTestModel作为文件名，然后单击 "添加"
4. 在 "实体数据模型向导" 页上，在 "选择模型内容" 对话框中选择 "空模型"
5. 单击"完成"

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。

该向导执行下列操作：

- 生成 EnumTestModel 文件，该文件定义概念模型、存储模型和这些模型之间的映射。设置要嵌入到输出程序集的 .edmx 文件的元数据项目处理属性，以便将生成的元数据文件嵌入到程序集中。

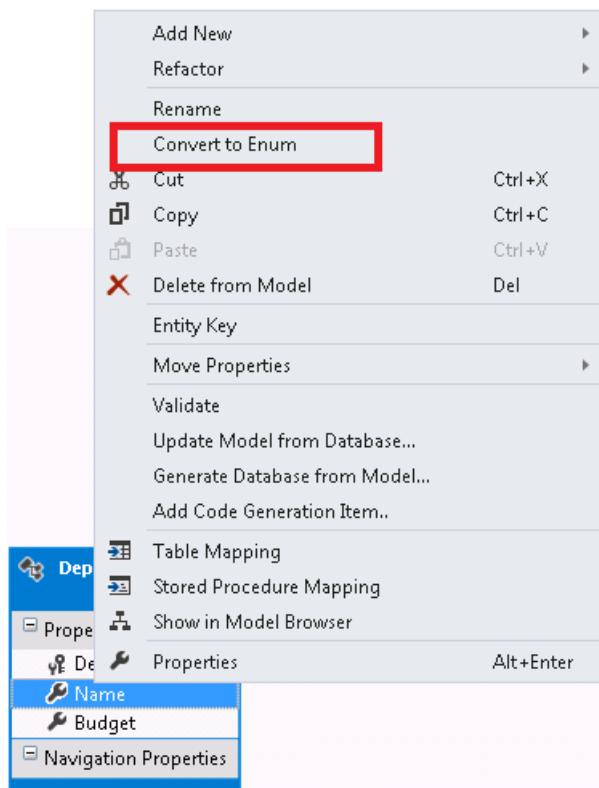
- 添加对以下程序集的引用: EntityFramework、System.componentmodel、DataAnnotations 和 System.object。
- 创建 EnumTestModel.tt 和 EnumTestModel.Context.tt 文件，并将它们添加到 .edmx 文件下。这些 T4 模板文件生成代码，该代码定义 DbContext 派生类型和映射到 .edmx 模型中的实体的 POCO 类型。

添加新的实体类型

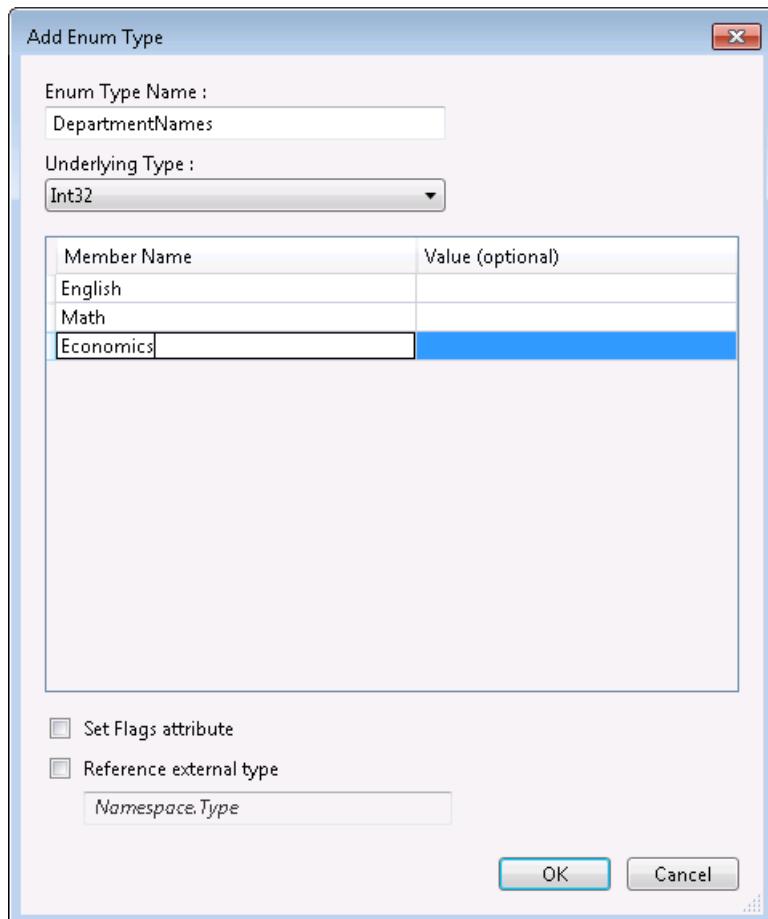
- 右键单击设计图面的空白区域，选择 “外接> 实体”，将显示 “新建实体” 对话框。
- 为类型名称指定部门并为键属性名称指定DepartmentID，将类型保留为Int32
- 单击“确定”
- 右键单击该实体，然后选择 “添加新的> 标量属性”
- 将新属性重命名为名称
- 将新属性的类型更改为Int32（默认情况下，新属性为字符串类型）若要更改类型，请打开属性窗口并将 type 属性更改为Int32
- 添加另一个标量属性，并将其重命名为预算，将类型更改为Decimal

添加枚举类型

- 在 Entity Framework Designer 中，右键单击 “名称” 属性，选择 “转换为枚举”



- 在 “添加枚举” 对话框中，键入DepartmentNames作为枚举类型名称，将基础类型更改为Int32，然后将以下成员添加到该类型中：英语、数学和经济性



3. 按 "确定"

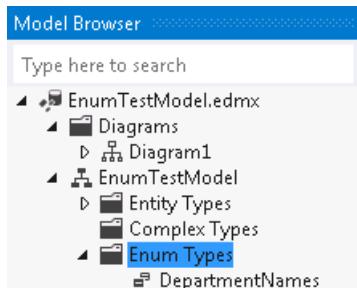
4. 保存模型并生成项目

NOTE

生成时，有关未映射实体和关联的警告可能出现在错误列表中。你可以忽略这些警告，因为在我们选择从模型生成数据库后，错误将消失。

如果你查看属性窗口，你会注意到，Name 属性的类型已更改为 **DepartmentNames**，并且新添加的枚举类型已添加到类型列表中。

如果切换到 "模型浏览器" 窗口，您将看到该类型也已添加到 "枚举类型" 节点。



NOTE

还可以通过单击鼠标右键并选择 "■"，从此窗口添加新枚举类型。一旦创建类型，该类型就会出现在类型列表中，并且你可以与属性关联。

从模型生成数据库

现在，我们可以生成一个基于该模型的数据库。

1. 右键单击 Entity Designer 图面上的空白区域，然后选择 "从模型生成数据库"
2. "生成数据库" 向导的 "选择您的数据连接" 对话框随即出现，单击 "新建连接" 按钮 "指定 (localdb)\mssqllocaldb 作为服务器名称，然后单击 "确定"
3. 询问是否要创建新数据库的对话框将弹出，单击 "是"。
4. 单击 "下一步"，"创建数据库向导" 生成用于创建数据库的数据定义语言(ddl)。在 "摘要和设置" 对话框中显示该 ddl 不包含映射到枚举类型的表的定义
5. 单击 "完成"，单击 "完成" 不执行 DDL 脚本。
6. 创建数据库向导执行以下操作：在 T-sql 编辑器中打开EnumTest生成存储架构并将连接字符串信息添加到 app.config 文件中的映射部分。
7. 在 T-sql 编辑器中单击鼠标右键，然后选择 "执行连接到服务器" 对话框，输入步骤2中的连接信息，然后单击 "连接"
8. 若要查看生成的架构，请在 SQL Server 对象资源管理器中右键单击数据库名称，然后选择 "刷新"

保留和检索数据

打开 Program.cs 文件，其中定义了 Main 方法。将以下代码添加到 Main 函数中。该代码将新的部门对象添加到上文下文中。然后，它会保存数据。此代码还执行 LINQ 查询，该查询返回名称为 DepartmentNames 的部门。

```
using (var context = new EnumTestModelContainer())
{
    context.Departments.Add(new Department{ Name = DepartmentNames.English });

    context.SaveChanges();

    var department = (from d in context.Departments
                      where d.Name == DepartmentNames.English
                      select d).FirstOrDefault();

    Console.WriteLine(
        "DepartmentID: {0} and Name: {1}",
        department.DepartmentID,
        department.Name);
}
```

编译并运行该应用程序。该程序生成以下输出：

```
DepartmentID: 1 Name: English
```

若要查看数据库中的数据，请在 SQL Server 对象资源管理器中右键单击数据库名称，然后选择 "刷新"。然后，单击表上的鼠标右键，然后选择 "查看数据"。

Summary

在本演练中，我们介绍了如何使用 Entity Framework Designer 映射枚举类型以及如何在代码中使用枚举。

空间-EF 设计器

2020/3/12 •

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

视频和分步演练演示了如何使用 Entity Framework Designer 映射空间类型。它还演示了如何使用 LINQ 查询查找两个位置之间的距离。

本演练将使用 Model First 创建新的数据库，但 EF 设计器也可以与[Database First](#)工作流一起用于映射到现有数据库。

实体框架5中引入了空间类型支持。请注意，若要使用空间类型、枚举和表值函数等新功能，则必须以 .NET Framework 4.5 为目标。默认情况下，Visual Studio 2012 面向 .NET 4.5。

若要使用空间数据类型，还必须使用具有空间支持的实体框架提供程序。有关详细信息，请参阅[提供程序对空间类型的支持](#)。

主要的空间数据类型有两种：地理和几何。Geography 数据类型存储椭圆体的数据（例如 GPS 纬度和经度坐标）。Geometry 数据类型表示欧氏（平面）坐标系。

观看视频

此视频演示如何用 Entity Framework Designer 映射空间类型。它还演示了如何使用 LINQ 查询查找两个位置之间的距离。

主讲人 :Julia Kornich

视频:[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

你将需要安装 Visual Studio 2012、旗舰版、高级版、专业版或 Web Express edition 才能完成此演练。

设置项目

1. 打开 Visual Studio 2012
2. 在 "文件" 菜单上，指向 "新建"，然后单击 "项目"
3. 在左窗格中，单击 "Visual C#"，然后选择控制台模板
4. 输入SpatialEFDDesigner作为项目名称，然后单击 "确定"

使用 EF 设计器创建新模型

1. 右键单击 "解决方案资源管理器" 中的项目名称，指向 "添加"，然后单击 "新建项"
2. 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"
3. 输入UniversityModel作为文件名，然后单击 "添加"
4. 在 "实体数据模型向导" 页上，在 "选择模型内容" 对话框中选择 "空模型"
5. 单击 "完成"

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。

该向导执行下列操作：

- 生成 EnumTestModel 文件，该文件定义概念模型、存储模型和这些模型之间的映射。设置要嵌入到输出程序集的 .edmx 文件的元数据项目处理属性，以便将生成的元数据文件嵌入到程序集中。
- 添加对以下程序集的引用：EntityFramework、System.ComponentModel、DataAnnotations 和 System.Object。
- 创建 UniversityModel.tt 和 UniversityModel.Context.tt 文件，并将它们添加到 .edmx 文件下。这些 T4 模板文件生成代码，该代码定义 DbContext 派生类型和映射到 .edmx 模型中的实体的 POCO 类型

添加新的实体类型

- 右键单击设计图面的空白区域，选择 “外接 > 实体”，将显示 “新建实体” 对话框。
- 指定类型名称的大学，并为键属性名称指定 UniversityID，将类型保留为 Int32
- 单击“确定”
- 右键单击该实体，然后选择 “添加新的 > 标量属性”
- 将新属性重命名为名称
- 添加另一个标量属性，并将其重命名为 Location 打开属性窗口并将新属性的类型更改为 Geography
- 保存模型并生成项目

NOTE

生成时，有关未映射实体和关联的警告可能出现在错误列表中。你可以忽略这些警告，因为在我们选择从模型生成数据库后，错误将消失。

从模型生成数据库

现在，我们可以生成一个基于该模型的数据库。

- 右键单击 Entity Designer 图面上的空白区域，然后选择 “从模型生成数据库”
- “生成数据库”向导的“选择您的数据连接”对话框随即出现，单击“新建连接”按钮“指定 (localdb)\mssqllocaldb 作为数据库的服务器名称和大学，并单击“确定”
- 询问是否要创建新数据库的对话框将弹出，单击“是”。
- 单击“下一步”，“创建数据库向导”生成用于创建数据库的数据定义语言 (DDL)。在“摘要和设置”对话框中显示该 DDL 不包含映射到枚举类型的表的定义
- 单击“完成”，单击“完成”不执行 DDL 脚本。
- 创建数据库向导执行以下操作：在 T-SQL 编辑器中打开 UniversityModel 生成存储架构并将连接字符串信息添加到 app.config 文件中的映射部分。
- 在 T-SQL 编辑器中单击鼠标右键，然后选择“执行连接到服务器”对话框，输入步骤 2 中的连接信息，然后单击“连接”
- 若要查看生成的架构，请在 SQL Server 对象资源管理器中右键单击数据库名称，然后选择“刷新”

保留和检索数据

打开 Program.cs 文件，其中定义了 Main 方法。将以下代码添加到 Main 函数中。

该代码将两个新的大学对象添加到上下文中。空间属性使用 DbGeography.FromText 方法进行初始化。将 WellKnownText 表示的地理点传递给方法。然后，该代码将保存数据。然后，将构造并执行 LINQ 查询，该查询返回其位置与指定位置最接近的大学对象。

```
using (var context = new UniversityModelContainer())
{
    context.Universities.Add(new University()
    {
        Name = "Graphic Design Institute",
        Location = DbGeography.FromText("POINT(-122.336106 47.605049)"),
    });

    context.Universities.Add(new University()
    {
        Name = "School of Fine Art",
        Location = DbGeography.FromText("POINT(-122.335197 47.646711)"),
    });

    context.SaveChanges();

    var myLocation = DbGeography.FromText("POINT(-122.296623 47.640405)");

    var university = (from u in context.Universities
                      orderby u.Location.Distance(myLocation)
                      select u).FirstOrDefault();

    Console.WriteLine(
        "The closest University to you is: {0}.",
        university.Name);
}
```

编译并运行该应用程序。该程序生成以下输出：

```
The closest University to you is: School of Fine Art.
```

若要查看数据库中的数据，请在 SQL Server 对象资源管理器中右键单击数据库名称，然后选择“刷新”。然后，单击表上的鼠标右键，然后选择“查看数据”。

Summary

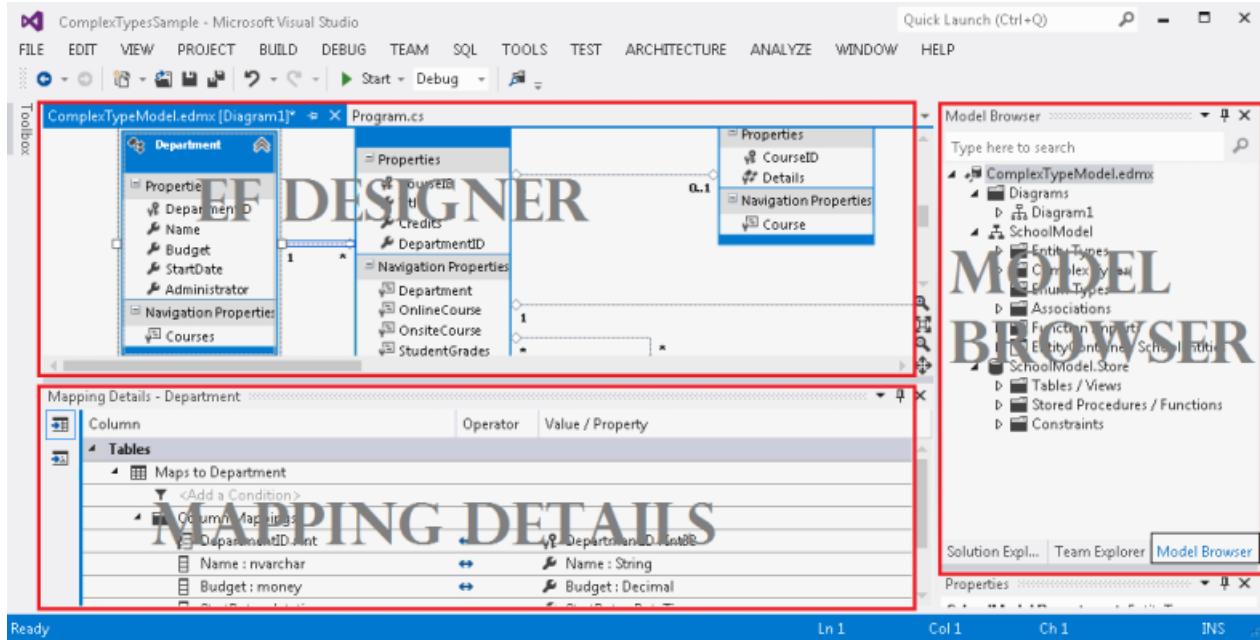
在本演练中，我们介绍了如何使用 Entity Framework Designer 映射空间类型，以及如何在代码中使用空间类型。

设计器实体拆分

2020/3/11 •

本演练演示如何通过使用 Entity Framework Designer (EF 设计器)修改模型，将一个实体类型映射到两个表。如果多个表共享同一个键，则可以将一个实体映射到这些表。将实体类型映射到两个表的概念可以轻松地扩展为将一个实体类型映射到两个以上的表。

下图显示了在使用 EF 设计器时使用的主窗口。



先决条件

Visual Studio 2012 或 Visual Studio 2010、旗舰版、高级版、专业版或 Web Express edition。

创建数据库

随 Visual Studio 一起安装的数据库服务器因安装的 Visual Studio 版本而异：

- 如果使用的是 Visual Studio 2012，则将创建一个 LocalDB 数据库。
- 如果使用的是 Visual Studio 2010，则将创建 SQL Express 数据库。

首先，我们将创建一个包含两个表的数据库，我们会将这些表合并到一个实体中。

- 打开 Visual Studio
- 视图-> 服务器资源管理器
- 右键单击 "数据连接-> 添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库，则需要选择 Microsoft SQL Server 作为数据源
- 连接到 LocalDB 或 SQL Express，具体取决于你安装的是哪个
- 输入 EntitySplitting 作为数据库名称
- 选择 "确定"，系统会询问您是否要创建新数据库，请选择 "是"
- 新数据库现在将出现在服务器资源管理器
- 如果使用的是 Visual Studio 2012
 - 在服务器资源管理器中右键单击该数据库，然后选择 "新建查询"

- 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择 "执行"
- 如果使用的是 Visual Studio 2010
 - 选择数据> Transact-sql 编辑器-> 新建查询连接 ...
 - 输入 。\SQLEXPRESS 作为服务器名称，然后单击 "确定"
 - 从 "查询编辑器" 顶部的下拉菜单中选择 " EntitySplitting " 数据库
 - 将以下 SQL 复制到新的查询中，然后右键单击该查询，然后选择 "执行 SQL "。

```

CREATE TABLE [dbo].[Person] (
[PersonId] INT IDENTITY (1, 1) NOT NULL,
[FirstName] NVARCHAR (200) NULL,
[LastName] NVARCHAR (200) NULL,
CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED ([PersonId] ASC)
);

CREATE TABLE [dbo].[PersonInfo] (
[PersonId] INT NOT NULL,
[Email] NVARCHAR (200) NULL,
[Phone] NVARCHAR (50) NULL,
CONSTRAINT [PK_PersonInfo] PRIMARY KEY CLUSTERED ([PersonId] ASC),
CONSTRAINT [FK_Person_PersonInfo] FOREIGN KEY ([PersonId]) REFERENCES [dbo].[Person] ([PersonId]) ON DELETE CASCADE
);

```

创建项目

- 在 "文件" 菜单上，指向 "新建"，再单击 "项目"。
- 在左窗格中，单击 "Visual C#"，然后选择 "控制台应用程序" 模板。
- 输入**MapEntityToTablesSample**作为项目名称，然后单击 "确定"。
- 如果系统提示保存在第一部分中创建的 SQL 查询，请单击 "否"。

创建基于数据库的模型

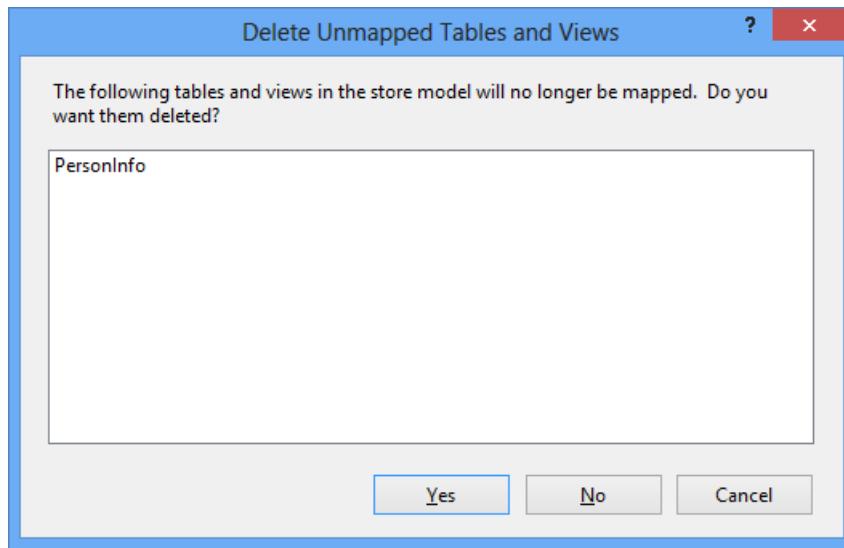
- 右键单击 "解决方案资源管理器" 中的项目名称，指向 "添加"，然后单击 "新建项"。
- 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"。
- 输入**MapEntityToTablesModel**作为文件名，然后单击 "添加"。
- 在 "选择模型内容" 对话框中，选择 "从数据库生成"，然后单击 "下一步"。
- 从下拉菜单中选择 "EntitySplitting" 连接，然后单击 "下一步"。
- 在 "选择数据库对象" 对话框中，选中 "表" 旁边的框 "节点。这会将**EntitySplitting**数据库中的所有表添加到模型。
- 单击 "完成"。

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。

将一个实体映射到两个表

在此步骤中，我们将更新**person**实体类型以合并**person**和**PersonInfo**表中的数据。

- 选择 **PersonInfo** 实体的 电子邮件 和 电话 属性，并按Ctrl + X键。
- 选择 "人员" 实体，并按Ctrl + V键。
- 在设计图面上，选择 **PersonInfo** 实体，并按键盘上的 "删除" 按钮。
- 当系统询问你是否要从模型中删除**PersonInfo**表时，请单击 "否"，我们将把它映射到**Person**实体。



接下来的步骤需要“窗口中的“映射详细信息”。如果看不到此窗口,请右键单击设计图面,然后选择“映射详细信息”。

- 选择“人员”实体类型,然后单击“映射详细信息”窗口中的<添加表或视图>。
- 从下拉列表中选择“**PersonInfo**”。“映射详细信息”窗口将更新为默认的列映射,这些映射适用于我们的方案。

实体类型的人员现在映射到和 PersonInfo 表。

A screenshot of the "Mapping Details - Person" window. It shows a table mapping between the Person entity type and the PersonInfo table. The table has columns: Column, Operator, and Value / Property. Under "Tables", there are two entries: "Maps to Person" and "Maps to PersonInfo". The "Maps to PersonInfo" entry is selected and highlighted with a blue bar. It contains column mappings for PersonId, FirstName, LastName, Email, and Phone.

使用模型

- 将以下代码粘贴到 Main 方法中。

```

using (var context = new EntitySplittingEntities())
{
    var person = new Person
    {
        FirstName = "John",
        LastName = "Doe",
        Email = "john@example.com",
        Phone = "555-555-5555"
    };

    context.People.Add(person);
    context.SaveChanges();

    foreach (var item in context.People)
    {
        Console.WriteLine(item.FirstName);
    }
}

```

- 编译并运行该应用程序。

由于运行此应用程序，对数据库执行了以下 T-sql 语句。

- 以下两个INSERT语句是执行上下文的结果。SaveChanges ()。它们采用person实体的数据，并将其拆分为person和PersonInfo表。

ADO.NET: Execute Reader "insert [dbo].[Person]([FirstName], [L
The command text "insert [dbo].[Person]([FirstName],
[LastName])
values (@0, @1)
select [PersonId]
from [dbo].[Person]
where @@ROWCOUNT > 0 and [PersonId] = scope_identity()
was executed on connection "data source=(localdb)
\v11.0;initial catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", building a SqlDataReader.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

ADO.NET: Execute NonQuery "insert [dbo].[PersonInfo]([Person
The command text "insert [dbo].[PersonInfo]([PersonId],
[Email], [Phone])
values (@0, @1, @2)
" was executed on connection "data source=(localdb)
\v11.0;initial catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", returning the number of rows affected.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

- 由于枚举数据库中的人员，因此执行了下面的SELECT。它将Person和PersonInfo表中的数据合并在一起。

ADO.NET: Execute Reader "SELECT [Extent1].[PersonId] AS [Per
The command text "SELECT
[Extent1].[PersonId] AS [PersonId],
[Extent2].[FirstName] AS [FirstName],
[Extent2].[LastName] AS [LastName],
[Extent1].[Email] AS [Email],
[Extent1].[Phone] AS [Phone]
FROM [dbo].[PersonInfo] AS [Extent1]
INNER JOIN [dbo].[Person] AS [Extent2] ON [Extent1].[PersonId] = [Extent2].[PersonId]" was executed on connection
"data source=(localdb)\v11.0;initial
catalog=EntitySplitting;integrated
security=True;MultipleActiveResultSets=True;App=EntityFram
ework", building a SqlDataReader.
Thread: Main Thread [2052]
Related views: [Locals](#) [Call Stack](#)

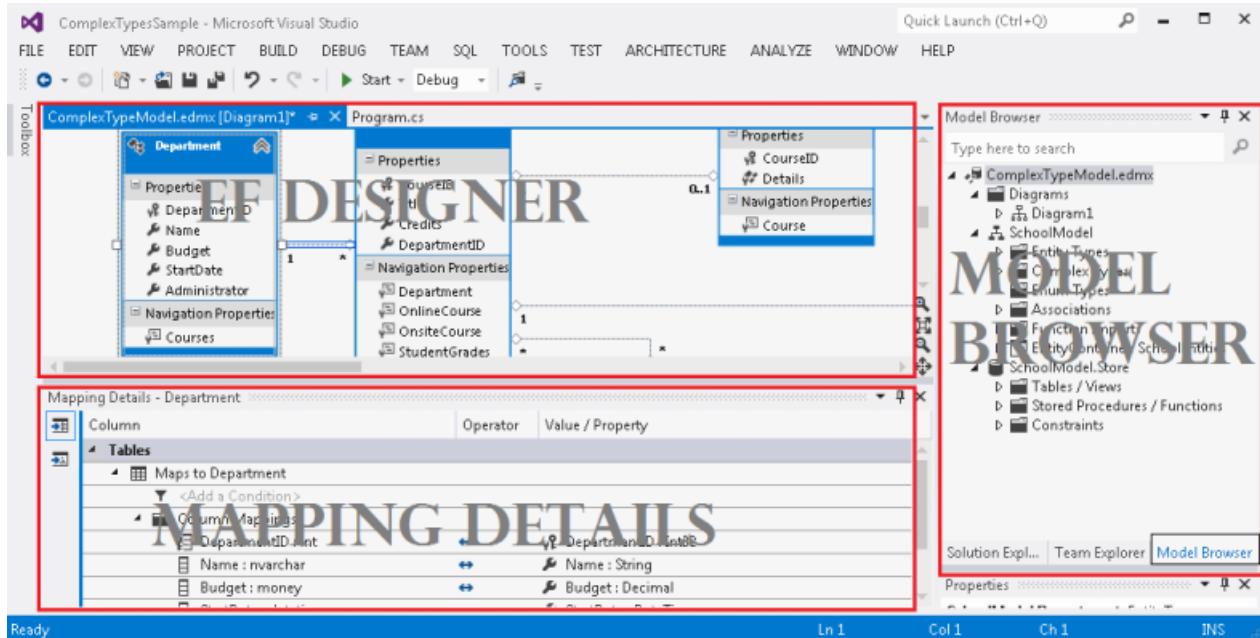
设计器表拆分

2020/3/11 •

本演练演示如何通过使用 Entity Framework Designer (EF 设计器)修改模型，将多个实体类型映射到单个表。

当使用延迟加载加载对象时，可能需要使用表拆分延迟加载某些属性。您可以将可能包含大量数据的属性分成单独的实体，并且仅在需要时加载。

下图显示了在使用 EF 设计器时使用的主窗口。



先决条件

若要完成此演练，您需要：

- Visual Studio 的最新版本。
- [School 示例数据库](#)。

设置项目

本演练使用 Visual Studio 2012。

- 打开 Visual Studio 2012。
- 在“文件”菜单上，指向“新建”，再单击“项目”。
- 在左窗格中，单击“Visual C#”，然后选择“控制台应用程序”模板。
- 输入TableSplittingSample作为项目名称，然后单击“确定”。

基于 School 数据库创建模型

- 右键单击“解决方案资源管理器”中的项目名称，指向“添加”，然后单击“新建项”。
- 从左侧菜单中选择“数据”，然后在“模板”窗格中选择“ADO.NET 实体数据模型”。
- 输入TableSplittingModel作为文件名，然后单击“添加”。
- 在“选择模型内容”对话框中，选择“从数据库生成”，然后单击“下一步”。
- 单击“新建连接”。在“连接属性”对话框中，输入服务器名称（例如，(localdb)\mssqllocaldb），选择身份验证

方法，为数据库名称键入 School，然后单击“确定”。“选择您的数据连接”对话框将通过数据库连接设置进行更新。

- 在“选择数据库对象”对话框中，展开“表”节点，然后检查 Person 表。这会将指定的表添加到 School 模型。
- 单击“完成”。

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。您在“选择数据库对象”对话框中选择的所有对象都将添加到模型中。

将两个实体映射到单个表

在本节中，您将把 Person 实体拆分为两个实体，然后将它们映射到一个表。

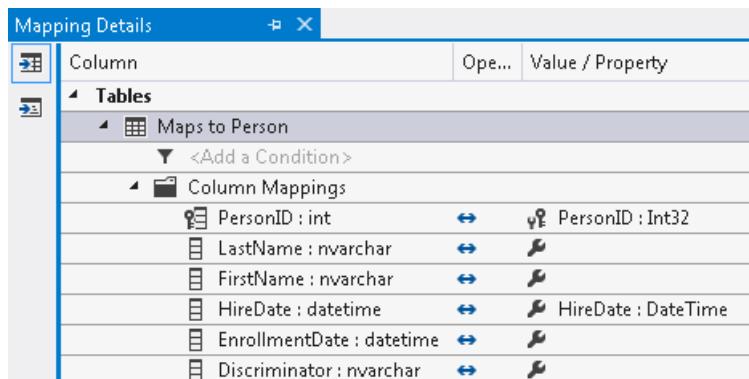
NOTE

Person 实体不包含任何可能包含大量数据的属性；它仅用作示例。

- 右键单击设计图面的空白区域，指向“添加新项”，然后单击“实体”。此时将显示“新建实体”对话框。
- 为“实体名称”和“PersonID”键入“HireInfo”作为键属性名称。
- 单击“确定”。
- 新的实体类型创建完毕，并且显示在设计图面上。
- 选择人员实体类型的“雇佣日期”属性，然后按 Ctrl + X 键。
- 选择 HireInfo 实体，并按 Ctrl + V 键。
- 创建 Person 和 HireInfo 之间的关联。为此，请右键单击设计图面的空白区域，指向“添加新项”，然后单击“关联”。
- 此时将显示“添加关联”对话框。默认情况下，指定 PersonHireInfo 名称。
- 指定关系两端的重数 1 (一)。
- 按“确定”。

下一步需要“窗口中的“映射详细信息”。如果看不到此窗口，请右键单击设计图面，然后选择“映射详细信息”。

- 选择 HireInfo 实体类型，然后在“映射详细信息”窗口中单击<添加表或视图>。
- 从<添加表或视图>字段“下拉列表中选择“人员”。此列表包含所选实体可以映射到的表或视图。默认情况下，应映射相应的属性。



- 在设计图面上选择“PersonHireInfo”关联。
- 右键单击设计图面上的关联，然后选择“属性”。
- 在“属性”窗口中，选择“引用约束”属性，然后单击省略号按钮。
- 从“主体”下拉列表中选择“人员”。
- 按“确定”。

使用模型

- 将以下代码粘贴到 Main 方法中。

```
using (var context = new SchoolEntities())
{
    Person person = new Person()
    {
        FirstName = "Kimberly",
        LastName = "Morgan",
        Discriminator = "Instructor",
    };

    person.HireInfo = new HireInfo()
    {
        HireDate = DateTime.Now
    };

    // Add the new person to the context.
    context.People.Add(person);

    // Insert a row into the Person table.
    context.SaveChanges();

    // Execute a query against the Person table.
    // The query returns columns that map to the Person entity.
    var existingPerson = context.People.FirstOrDefault();

    // Execute a query against the Person table.
    // The query returns columns that map to the Instructor entity.
    var hireInfo = existingPerson.HireInfo;

    Console.WriteLine("{0} was hired on {1}",
        existingPerson.LastName, hireInfo.HireDate);
}
```

- 编译并运行该应用程序。

由于运行此应用程序，以下 T-sql 语句针对 School 数据库执行。

- 执行上下文时，执行了以下插入操作。SaveChanges () 并结合 Person 和 HireInfo 实体中的数据

⚡ ADO.NET: Execute Reader "insert [dbo].[Person]([LastName], [FirstName], [HireDate], [EnrollmentDate], [Discriminator]) values (@0, @1, @2, null, @3) select [PersonID] from [dbo].[Person] where @@ROWCOUNT > 0 and [PersonID] = scope_identity()" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.

- 执行上下文时，执行了以下 SELECT 。FirstOrDefault () 并只选择映射到人员的列

⚡ ADO.NET: Execute Reader "SELECT TOP (1) [c].[PersonID], [c].[PersonID] AS [PersonID], [c].[LastName] AS [LastName], [c].[FirstName] AS [FirstName], [c].[EnrollmentDate] AS [EnrollmentDate], [c].[Discriminator] AS [Discriminator] FROM [dbo].[Person] AS [c]" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.

- 在访问导航属性 existingPerson 的结果中执行了下面的SELECT语句，并只选择了映射到HireInfo的列

⚡ ADO.NET: Execute Reader "SELECT [Extent1].[PersonID], [Extent1].[PersonID] AS [PersonID], [Extent1].[HireDate] AS [HireDate] FROM [dbo].[Person] AS [Extent1] WHERE [Extent1].[PersonID] = @EntityKeyValue1" was executed on connection "data source=(localdb)\v11.0;initial catalog=School;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework", building a SqlDataReader.

设计器 TPH 继承

2020/3/11 ·

此分步演练演示如何使用 Entity Framework Designer (EF 设计器) 在概念模型中实现每个层次结构一个表 (TPH) 继承。TPH 继承使用一个数据库表来维护继承层次结构中所有实体类型的数据。

在本演练中，我们会将 Person 表映射到三个实体类型：Person (基类型)、学生 (派生自人员) 和讲师 (派生自人员)。我们将从数据库 (Database First) 创建概念模型，然后使用 EF 设计器更改模型以实现 TPH 继承。

可以使用 Model First 映射到 TPH 继承，但您必须编写自己的数据库生成工作流，该工作流非常复杂。然后，将此工作流分配到 EF 设计器中的 "数据库生成工作流" 属性。更简单的替代方法是使用 Code First。

其他继承选项

每种类型一个表 (TPT) 是另一种类型的继承，其中，数据库中的单独表映射到参与继承的实体。有关如何将每种类型一个表继承映射到 EF 设计器的信息，请参阅[Ef 设计器 TPT 继承](#)。

实体框架运行时支持每个具体的表类型继承 (TPC) 和混合继承模型，但 EF 设计器不支持。如果要使用 TPC 或混合继承，则有两个选项：使用 Code First 或手动编辑 EDMX 文件。如果选择处理 EDMX 文件，“映射详细信息”窗口将进入“安全模式”，您将无法使用设计器来更改映射。

先决条件

若要完成此演练，您需要：

- Visual Studio 的最新版本。
- [School 示例数据库](#)。

设置项目

- 打开 Visual Studio 2012。
- 选择 "文件->"> 项目
- 在左窗格中，单击 "Visual C#"，然后选择 "控制台" 模板。
- 输入 "TPHDBFirstSample" 作为名称。
- 选择"确定"。****

创建模型

- 右键单击 "解决方案资源管理器" 中的项目名称，然后选择"添加-> 新项"。
- 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"。
- 输入TPHModel作为文件名，然后单击 "添加"。
- 在 "选择模型内容" 对话框中，选择 "从数据库生成"，然后单击 "下一步"。
- 单击 "新建连接"。在 "连接属性" 对话框中，输入服务器名称(例如，(localdb)\mssqllocaldb)，选择身份验证方法，为数据库名称键入 School，然后单击 "确定"。"选择您的数据连接" 对话框将通过数据库连接设置进行更新。
- 在 "选择数据库对象" 对话框中的 "表" 节点下，选择 Person 表。
- 单击 "完成"。

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。您在 "选择数据库对象" 对话框中选择的所有对象都将添加到模型中。

这就是 "Person" 表在数据库中的外观。

The screenshot shows the EntityDataSource1 Designer window with the title bar 'dbo.Person [Design]'. Below the title bar are two tabs: 'Update' and 'Script File: dbo.Person.sql'. The main area is a table with four columns: 'Name', 'Data Type', 'Allow Nulls', and a small checkbox column. The table contains six rows, each representing a column in the Person table:

Name	Data Type	Allow Nulls	
PersonID	int	<input type="checkbox"/>	
LastName	nvarchar(50)	<input type="checkbox"/>	
FirstName	nvarchar(50)	<input type="checkbox"/>	
HireDate	datetime	<input checked="" type="checkbox"/>	
EnrollmentDate	datetime	<input checked="" type="checkbox"/>	
Discriminator	nvarchar(50)	<input type="checkbox"/>	
		<input type="checkbox"/>	

实现每个层次结构一个表继承

Person表具有鉴别器列，该列可以具有以下两个值之一：“Student”和“讲师”。根据值，Person表将映射到“学生”实体或“讲师”实体。Person表还具有两列：“雇佣日期”和“EnrollmentDate”，这些列必须可以为null，因为用户不能同时是学生和指导员（至少在本演练中没有）。

添加新实体

- 添加新实体。为此，请右键单击 Entity Framework Designer 设计图面的空白区域，然后选择“>”实体。
- 为“实体名称”键入“指导员”然后从“基类型”下拉列表中选择“人员”。
- 单击“确定”。
- 添加另一个新实体。为“实体名称”键入“Student”然后从“基类型”下拉列表中选择“人员”。

已将两个新的实体类型添加到设计图面。箭头从新的实体类型指向 实体类型的 人员;这表示 用户 是新实体类型的基类型。

- 右键单击 人员 实体的“雇佣日期”属性。选择“剪切”（或使用 Ctrl + X 键）。
- 右键单击 讲师 实体，然后选择“粘贴”（或使用 Ctrl + V 键）。
- 右键单击“雇佣日期”属性，然后选择“属性”。
- 在“属性”窗口中，将“可以为 null”的属性设置为 false。
- 右键单击 Person 实体的 EnrollmentDate 属性。选择“剪切”（或使用 Ctrl + X 键）。
- 右键单击“学生”实体，然后选择“粘贴”（或使用 Ctrl + V 键）。
- 选择 EnrollmentDate 属性，并将可为 null 的属性设置为 false。
- 选择“人员”实体类型。在“属性”窗口中，将其 Abstract 属性设置为 true。
- 删除Person的鉴别器属性。下一节将对其删除原因进行说明。

映射实体

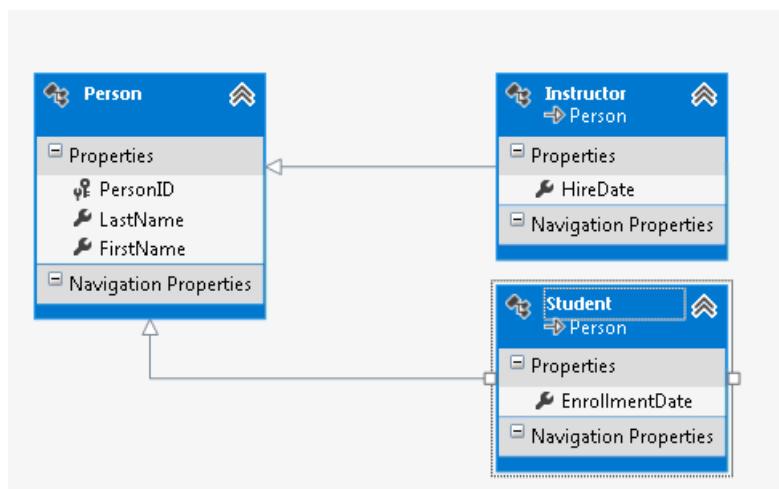
- 右键单击 讲师，然后选择“表映射”。在“映射详细信息”窗口中选择“指导员”实体。
- 在“映射详细信息”窗口中，单击“<添加表或视图>”。<添加表或视图>“字段将成为所选实体可映射到的表或视图的下拉列表。
- 从下拉列表中选择 Person。
- “映射详细信息”窗口使用默认列映射和一个用于添加条件的选项进行更新。
- 单击“<>添加条件”。<“添加条件”>“字段将成为可以设置条件的列的下拉列表。
- 从下拉列表中选择“鉴别器”。
- 在“映射详细信息”窗口的“运算符”列中，从下拉列表中选择“=”。
- 在“值/属性”列中，键入“讲师”。最终结果应该如下所示：

Column	Operator	Value / Property
Tables		
Maps to Person	=	Instructor
When Discriminator	=	Instructor
<Add a Condition>	=	Instructor
Column Mappings		
HireDate : datetime	↔	HireDate : DateTime
EnrollmentDate : datetime	↔	
Discriminator : nvarchar	↔	

- 为 "学生" 实体型重复上述步骤，但将条件设置为 "学生" 值。

删除鉴别器属性的原因是，您不能多次映射一个表列。此列将用于条件映射，因此它也不能用于属性映射。对于这两种情况，唯一的方法是：如果条件使用 `Is null` 或者不是 `null` 比较。

"每个层次结构一个表" 继承实现完毕。



使用模型

打开 `Program.cs` 文件，其中定义了 `Main` 方法。将以下代码粘贴到 `Main` 函数中。此代码执行三个查询。第一个查询返回所有 `Person` 对象。第二个查询使用 `OfType` 方法返回指导员对象。第三个查询使用 `OfType` 方法返回学生对象。

```

using (var context = new SchoolEntities())
{
    Console.WriteLine("All people:");
    foreach (var person in context.People)
    {
        Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
    }

    Console.WriteLine("Instructors only: ");
    foreach (var person in context.People.OfType<Instructor>())
    {
        Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
    }

    Console.WriteLine("Students only: ");
    foreach (var person in context.People.OfType<Student>())
    {
        Console.WriteLine("{0} {1}", person.FirstName, person.LastName);
    }
}
  
```

设计器 TPT 继承

2020/3/11 •

此分步演练演示如何使用 Entity Framework Designer (EF 设计器) 实现模型中的每种类型一个表(TPT)继承。每种类型一个表继承使用数据库中单独的表为继承层次结构中的每种类型维护非继承属性和键属性的数据。

在本演练中，我们会将课程(基类型)、`OnlineCourse`(派生自课程)和为 `onsitecourse`(从课程派生)到同名的表。我们将从数据库创建一个模型，然后更改模型以实现 TPT 继承。

还可以从 Model First 开始，然后从模型生成数据库。默认情况下，EF 设计器会使用 TPT 策略，因此该模型中的任何继承都将映射到单独的表。

其他继承选项

每个层次结构一个表(TPH)是另一种类型的继承，其中，一个数据库表用于维护继承层次结构中所有实体类型的数据。有关如何用 Entity Designer 映射每个层次结构一个表继承的信息，请参阅[EF 设计器 TPH 继承](#)。

请注意，实体框架运行时支持每个具体的表类型继承(TPC)和混合继承模型，但 EF 设计器不支持。如果要使用 TPC 或混合继承，则有两个选项：使用 Code First 或手动编辑 EDMX 文件。如果选择处理 EDMX 文件，“映射详细信息”窗口将进入“安全模式”，您将无法使用设计器来更改映射。

先决条件

若要完成此演练，您需要：

- Visual Studio 的最新版本。
- [School 示例数据库](#)。

设置项目

- 打开 Visual Studio 2012。
- 选择“文件->”> 项目
- 在左窗格中，单击“Visual C#”，然后选择“控制台”模板。
- 输入“TPTDBFirstSample”作为名称。
- 选择“确定”。****

创建模型

- 右键单击“解决方案资源管理器”中的项目，然后选择“添加-> 新项”。
- 从左侧菜单中选择“数据”，然后在“模板”窗格中选择“ADO.NET 实体数据模型”。
- 输入 TPTModel 作为文件名，然后单击“添加”。
- 在“选择模型内容”对话框中，选择“** 从数据库生成**”，然后单击“下一步”。
- 单击“新建连接”。在“连接属性”对话框中，输入服务器名称（例如，`(localdb)\mssqllocaldb`），选择身份验证方法，为数据库名称键入 School，然后单击“确定”。“选择您的数据连接”对话框将通过数据库连接设置进行更新。
- 在“选择数据库对象”对话框的“表”节点下，选择“部门”、“课程”、“`OnlineCourse`”和“为 `onsitecourse`”表。
- 单击“完成”。

此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。您在“选择数据库对象”对话框中选择的所有对

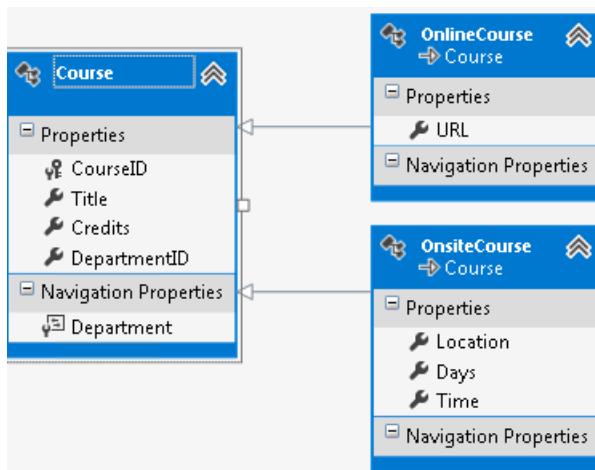
象都将添加到模型中。

实现每种类型一个表继承

- 在设计图面上，右键单击**OnlineCourse**实体类型，然后选择“属性”。
- 在“属性”窗口中，将“基类型”属性设置为“课程”。
- 右键单击**onsitecourse**实体类型，然后选择“属性”。
- 在“属性”窗口中，将“基类型”属性设置为“课程”。
- 右键单击**OnlineCourse**和课程实体类型之间的关联(行)。选择“从模型删除”。
- 右键单击**onsitecourse**和课程实体类型之间的关联。选择“从模型删除”。

现在，我们将从**OnlineCourse**和**onsitecourse**中删除**CourseID**属性，因为这些类从课程基类型继承**CourseID**。

- 右键单击**OnlineCourse**实体类型的**CourseID**属性，然后选择“从模型中删除”。
- 右键单击**onsitecourse**实体类型的**CourseID**属性，然后选择“从模型删除”
- 至此已实现每种类型一个表继承。



使用模型

打开**Program.cs**文件，其中定义了**Main**方法。将以下代码粘贴到**Main**函数中。此代码执行三个查询。第一个查询返回与指定部门相关的所有课程。第二个查询使用**OfType**方法返回与指定部门相关的**OnlineCourses**。第三个查询返回**OnsiteCourses**。

```
using (var context = new SchoolEntities())
{
    foreach (var department in context.Departments)
    {
        Console.WriteLine("The {0} department has the following courses:",
                          department.Name);

        Console.WriteLine("    All courses");
        foreach (var course in department.Courses )
        {
            Console.WriteLine("        {0}", course.Title);
        }

        foreach (var course in department.Courses.
                  OfType<OnlineCourse>())
        {
            Console.WriteLine("        Online - {0}", course.Title);
        }

        foreach (var course in department.Courses.
                  OfType<OnsiteCourse>())
        {
            Console.WriteLine("        Onsite - {0}", course.Title);
        }
    }
}
```

设计器查询存储过程

2020/3/11 ·

此分步演练演示如何使用 Entity Framework Designer (EF 设计器) 将存储过程导入到模型中，然后调用导入的存储过程来检索结果。

请注意，Code First 不支持映射到存储过程或函数。但是，可以使用 `DbSet.SqlQuery` 方法调用存储过程或函数。例如：

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]");
```

先决条件

若要完成此演练，您需要：

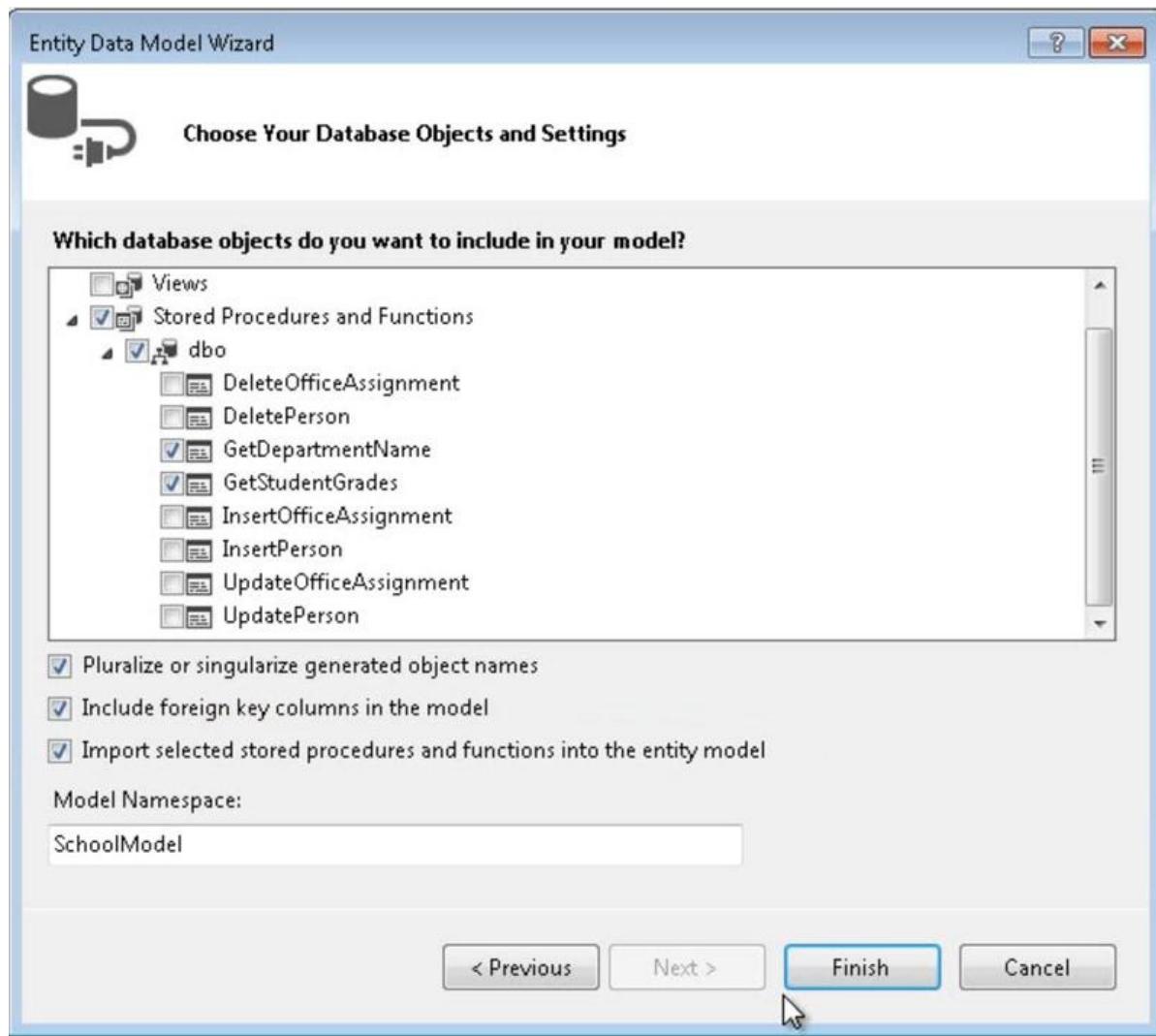
- Visual Studio 的最新版本。
- [School 示例数据库](#)。

设置项目

- 打开 Visual Studio 2012。
- 选择 "文件->"> 项目
- 在左窗格中，单击 "Visual C#"，然后选择 "控制台" 模板。
- 输入 "EFwithSProcsSample" 作为名称。
- 选择"确定"。****

创建模型

- 在解决方案资源管理器中右键单击项目，然后选择 "添加-> 新项"。
- 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"。
- 输入EFwithSProcsModel作为文件名，然后单击 "添加"。
- 在 "选择模型内容" 对话框中，选择 "从数据库生成"，然后单击 "下一步"。
- 单击 "新建连接"。
在 "连接属性" 对话框中，输入服务器名称(例如，`(localdb)\mssqllocaldb`)，选择身份验证方法，为数据库名称键入 School，然后单击 "确定"。
"选择您的数据连接" 对话框将通过数据库连接设置进行更新。
- 在 "选择数据库对象" 对话框中，选中 "表" 复选框以选择所有表。
另外，在 "存储过程和函数" 节点下选择以下存储过程：GetStudentGrades 和 GetDepartmentName。



从 Visual Studio 2012 开始，EF 设计器支持存储过程的大容量导入。默认情况下，将选中“将选定的存储过程和函数导入 entity 模型”。

- 单击“完成”。

默认情况下，返回多个列的每个导入的存储过程或函数的结果形状将自动成为新的复杂类型。在此示例中，我们想要将 GetStudentGrades 函数的结果映射到 StudentGrade 实体，并将 GetDepartmentName 结果映射到“无”（默认值为“无”）。

为了使函数导入返回实体类型，由相应的存储过程返回的列必须与返回的实体类型的标量属性完全匹配。函数导入还可以返回简单类型、复杂类型或无值的集合。

- 右键单击设计图面，然后选择“模型浏览器”。
- 在模型浏览器中，选择“函数导入”，然后双击“GetStudentGrades”函数。
- 在“编辑函数导入”对话框中，选择“实体”然后选择“StudentGrade”。

函数导入对话框顶部的“函数导入是可组合的”复选框允许您映射到可组合函数。如果选中此复选框，则“存储过程/函数名称”下拉列表中将只显示可组合函数（表值函数）。如果不选中此框，则列表中将只显示不可组合的函数。

使用模型

打开 Program.cs 文件，其中定义了 Main 方法。将以下代码添加到 Main 函数中。

此代码将调用两个存储过程：GetStudentGrades（为指定的 StudentId 返回 StudentGrades）和 GetDepartmentName（返回 output 参数中部门的名称）。

```
using (SchoolEntities context = new SchoolEntities())
{
    // Specify the Student ID.
    int studentId = 2;

    // Call GetStudentGrades and iterate through the returned collection.
    foreach (StudentGrade grade in context.GetStudentGrades(studentId))
    {
        Console.WriteLine("StudentID: {0}\tSubject={1}", studentId, grade.Subject);
        Console.WriteLine("Student grade: " + grade.Grade);
    }

    // Call GetDepartmentName.
    // Declare the name variable that will contain the value returned by the output parameter.
    ObjectParameter name = new ObjectParameter("Name", typeof(String));
    context.GetDepartmentName(1, name);
    Console.WriteLine("The department name is {0}", name.Value);

}
```

编译并运行该应用程序。该程序生成以下输出：

```
StudentID: 2
Student grade: 4.00
StudentID: 2
Student grade: 3.50
The department name is Engineering
```

输出参数

如果使用了 output 参数，则在完全读取结果之前，它们的值将不可用。这是因为 DbDataReader 的基础行为，有关详细信息，请参阅[使用 DataReader 检索数据](#)。

设计器 CUD 存储过程

2020/3/11 ·

此分步演练演示如何使用 Entity Framework Designer (EF 设计器) 将实体类型的 create\插入、更新和删除(CUD)操作映射到存储过程。默认情况下，实体框架会自动为 CUD 操作生成 SQL 语句，但也可以将存储过程映射到这些操作。

请注意，Code First 不支持映射到存储过程或函数。但是，可以使用 DbSet.SqlQuery 方法调用存储过程或函数。例如：

```
var query = context.Products.SqlQuery("EXECUTE [dbo].[GetAllProducts]");
```

将 CUD 操作映射到存储过程时的注意事项

将 CUD 操作映射到存储过程时，请注意以下事项：

- 如果要将其中一个 CUD 操作映射到存储过程，请映射所有这些操作。如果未映射所有这三个映射，则未映射的操作将失败(如果执行)并引发 `UpdateException`。
- 必须将存储过程的每个参数映射到实体属性。
- 如果服务器为插入的行生成主键值，则必须将此值映射回实体的键属性。在下面的示例中，`InsertPerson` 存储过程返回新创建的主键作为存储过程的结果集的一部分。使用>EF 设计器 功能 <添加结果绑定，将主键映射到实体键(`PersonID`)。
- 存储过程调用映射到概念模型中的实体1:1。例如，如果在概念模型中实现了继承层次结构，然后映射父级(base)和子(派生)实体的 CUD 存储过程，则保存子更改将仅调用子对象的存储过程，而不会触发父级的存储过程调用。

先决条件

若要完成此演练，您需要：

- Visual Studio 的最新版本。
- [School 示例数据库](#)。

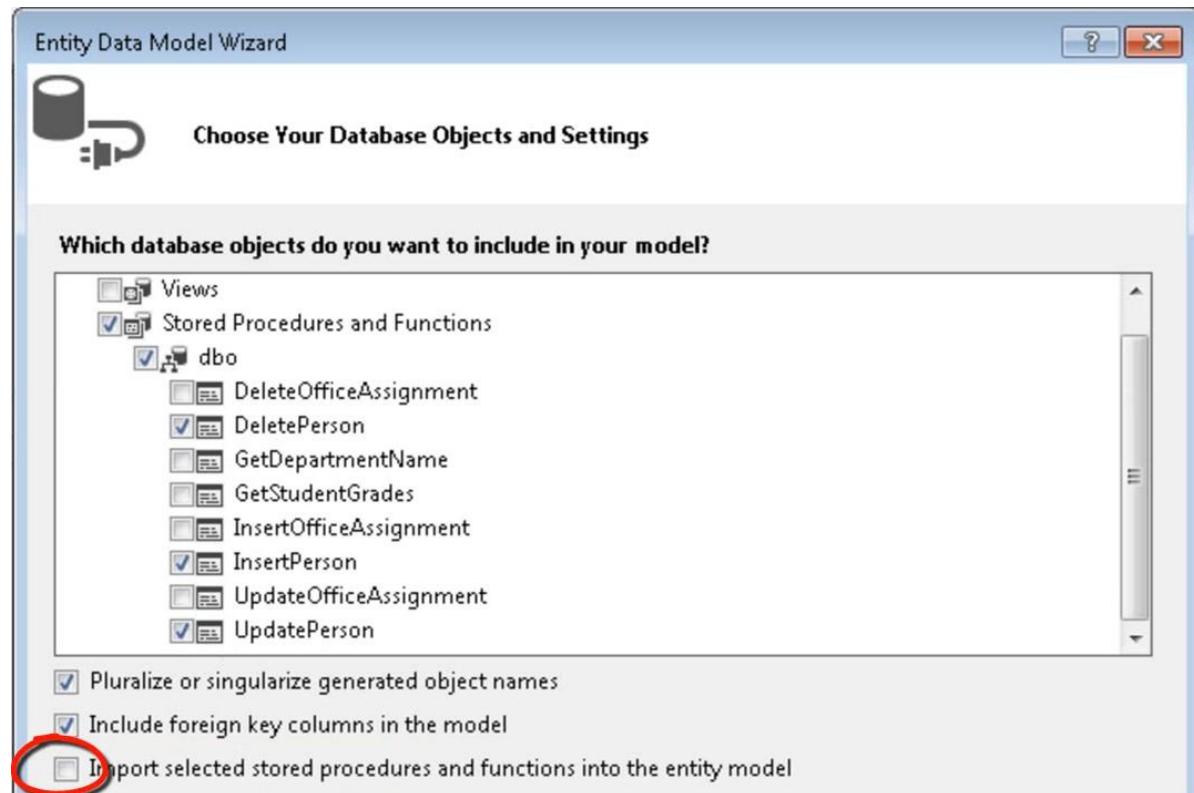
设置项目

- 打开 Visual Studio 2012。
- 选择 "文件->"> 项目
- 在左窗格中，单击 "Visual C#"，然后选择 "控制台" 模板。
- 输入 "`CUDSProcsSample`" 作为名称。
- 选择"确定"。****

创建模型

- 右键单击 "解决方案资源管理器" 中的项目名称，然后选择"添加-> 新项"。
- 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"。
- 输入`CUDSProcs`作为文件名，然后单击 "添加"。
- 在 "选择模型内容" 对话框中，选择 "从数据库生成"，然后单击 "下一步"。

- 单击“新建连接”。在“连接属性”对话框中，输入服务器名称（例如，(localdb)\mssqllocaldb），选择身份验证方法，为数据库名称键入 School，然后单击“确定”。“选择您的数据连接”对话框将通过数据库连接设置进行更新。
- 在“选择数据库对象”对话框中的“表”节点下，选择 Person 表。
- 另外，在“存储过程和函数”节点下选择以下存储过程：DeletePerson、InsertPerson 和 UpdatePerson。
- 从 Visual Studio 2012 开始，EF 设计器支持存储过程的大容量导入。默认情况下，将选中“将选定的存储过程和函数导入实体模型”。由于在此示例中，我们已存储了用于插入、更新和删除实体类型的存储过程，因此我们不想将它们导入，并将取消选中此复选框。



- 单击“完成”。显示了 EF 设计器（提供编辑模型的设计图面）。

将 Person 实体映射到存储过程

- 右键单击“人员”实体类型，然后选择“存储过程映射”。
- 存储过程映射显示在“映射详细信息”窗口中。
- 单击<选择“插入函数”>。该字段变成一个下拉列表，该列表中包含存储模型中可以映射到概念模型中的实体类型的存储过程。从下拉列表中选择“InsertPerson”。
- 此时将显示存储过程参数和实体属性之间的默认映射。请注意，箭头指示映射方向：向存储过程参数提供属性值。
- 单击<添加结果绑定>。
- 键入 NewPersonID，InsertPerson 存储过程返回的参数的名称。请确保不要键入前导空格或尾随空格。
- 按 enter。
- 默认情况下，NewPersonID 映射到实体键 PersonID。请注意，箭头指示映射的方向：向属性提供结果列的值。

Parameter / Column	Operator	Property	Use Original...	Rows Affected Parameter
Functions				
Insert Using InsertPerson				
Parameters				
@ LastName : nvarchar	←	LastName : String	<input type="checkbox"/>	
@ FirstName : nvarchar	←	FirstName : String	<input type="checkbox"/>	
@ HireDate : datetime	←	HireDate : DateTime	<input type="checkbox"/>	
@ EnrollmentDate : datetime	←	EnrollmentDate : DateTime	<input type="checkbox"/>	
@ Discriminator : nvarchar	←	Discriminator : String	<input type="checkbox"/>	
Result Column Bindings				
NewPersonID	→	PersonID : Int32		

- 单击“<> 选择”更新函数，然后从生成的下拉列表中选择“**UpdatePerson**”。
- 此时将显示存储过程参数和实体属性之间的默认映射。
- 单击“<> 选择”删除函数，然后从生成的下拉列表中选择“**DeletePerson**”。
- 此时将显示存储过程参数和实体属性之间的默认映射。

用户实体类型的插入、更新和删除操作现已映射到存储过程。

如果要在使用存储过程更新或删除实体时启用并发检查，请使用下列选项之一：

- 使用OUTPUT参数从存储过程中返回受影响的行数，并选中参数名称旁的“受影响的行”参数复选框。如果调用操作时返回的值为零，则将引发 **OptimisticConcurrencyException**。
- 选中要用于并发检查的属性旁边的“使用原始值”复选框。尝试更新时，在将数据写回数据库时，将使用最初从数据库读取的属性的值。如果值与数据库中的值不匹配，则将引发**OptimisticConcurrencyException**。

使用模型

打开Program.cs文件，其中定义了Main方法。将以下代码添加到 Main 函数中。

此代码创建一个新的Person对象，然后更新该对象，最后删除该对象。

```

using (var context = new SchoolEntities())
{
    var newInstructor = new Person
    {
        FirstName = "Robyn",
        LastName = "Martin",
        HireDate = DateTime.Now,
        Discriminator = "Instructor"
    }

    // Add the new object to the context.
    context.People.Add(newInstructor);

    Console.WriteLine("Added {0} {1} to the context.",
        newInstructor.FirstName, newInstructor.LastName);

    Console.WriteLine("Before SaveChanges, the PersonID is: {0}",
        newInstructor.PersonID);

    // SaveChanges will call the InsertPerson sproc.
    // The PersonID property will be assigned the value
    // returned by the sproc.
    context.SaveChanges();

    Console.WriteLine("After SaveChanges, the PersonID is: {0}",
        newInstructor.PersonID);

    // Modify the object and call SaveChanges.
    // This time, the UpdatePerson will be called.
    newInstructor.FirstName = "Rachel";
    context.SaveChanges();

    // Remove the object from the context and call SaveChanges.
    // The DeletePerson sproc will be called.
    context.People.Remove(newInstructor);
    context.SaveChanges();

    Person deletedInstructor = context.People.
        Where(p => p.PersonID == newInstructor.PersonID).
        FirstOrDefault();

    if (deletedInstructor == null)
        Console.WriteLine("A person with PersonID {0} was deleted.",
            newInstructor.PersonID);
}

```

- 编译并运行该应用程序。该程序生成以下输出 *

NOTE

PersonID 是由服务器自动生成的，因此很可能会看到不同的数字 *

```

Added Robyn Martin to the context.
Before SaveChanges, the PersonID is: 0
After SaveChanges, the PersonID is: 51
A person with PersonID 51 was deleted.

```

如果使用的是 Visual Studio 的旗舰版，则可以将 Intellitrace 与调试器结合使用，以查看执行的 SQL 语句。

IntelliTrace

Video: How to use Intellitrace

All Categories All Threads

Search

Debugger: Beginning of Application: Main, Program.cs

XML: XmlDocument Loaded

Debugger: Breakpoint Hit: Main, Program.cs line 33

ADO.NET: Execute Reader "[dbo].[InsertPerson]"

Debugger: Step Recorded: Main, Program.cs line 35

ADO.NET: Execute NonQuery "[dbo].[UpdatePerson]"

ADO.NET: Execute NonQuery "[dbo].[DeletePerson]"

ADO.NET: Execute Reader "SELECT TOP (1) [Extent1].[I

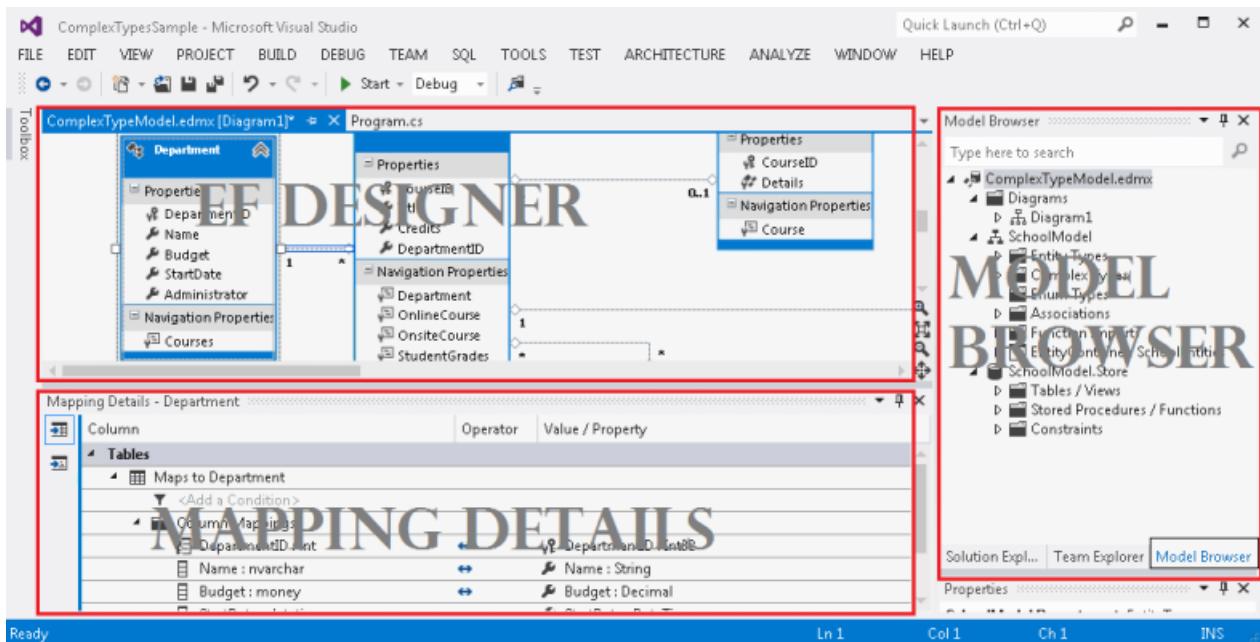
关系-EF 设计器

2020/3/11 ·

NOTE

本页提供有关使用 EF 设计器设置模型中的关系的信息。有关 EF 中的关系以及如何使用关系访问和操作数据的一般信息，请参阅[关系 & 导航属性](#)。

关联定义模型中的实体类型之间的关系。本主题演示如何映射与 Entity Framework Designer (EF 设计器) 的关联。下图显示了在使用 EF 设计器时使用的主窗口。



NOTE

在生成概念模型时，有关未映射的实体和关联的警告可能会显示在“错误列表”中。您可以忽略这些警告，因为在您选择从模型生成数据库后，错误将消失。

关联概述

使用 EF 设计器设计模型时，.edmx 文件表示您的模型。在 .edmx 文件中，**Association** 元素定义了两个实体类型之间的关系。关联必须指定关系中涉及的实体类型和关系的每一端可能的实体类型数量(也称为重数)。关联端的多重性值可以为一(1)、零或一(0 ..1)或多(*)。此信息是在两个子结束元素中指定的。

在运行时，可以通过导航属性或外键(如果在实体中选择公开外键)来访问关联一端的实体类型实例。公开外键后，将使用**ReferentialConstraint**元素(**Association**元素的子元素)管理实体之间的关系。建议你始终公开实体中关系的外键。

NOTE

在多对多(*:*)中，不能向实体中添加外键。在 *: * 关系中，使用独立对象管理关联信息。

有关 CSDL 元素(**ReferentialConstraint**、**关联**等)的信息，请参阅[csdl 规范](#)。

创建和删除关联

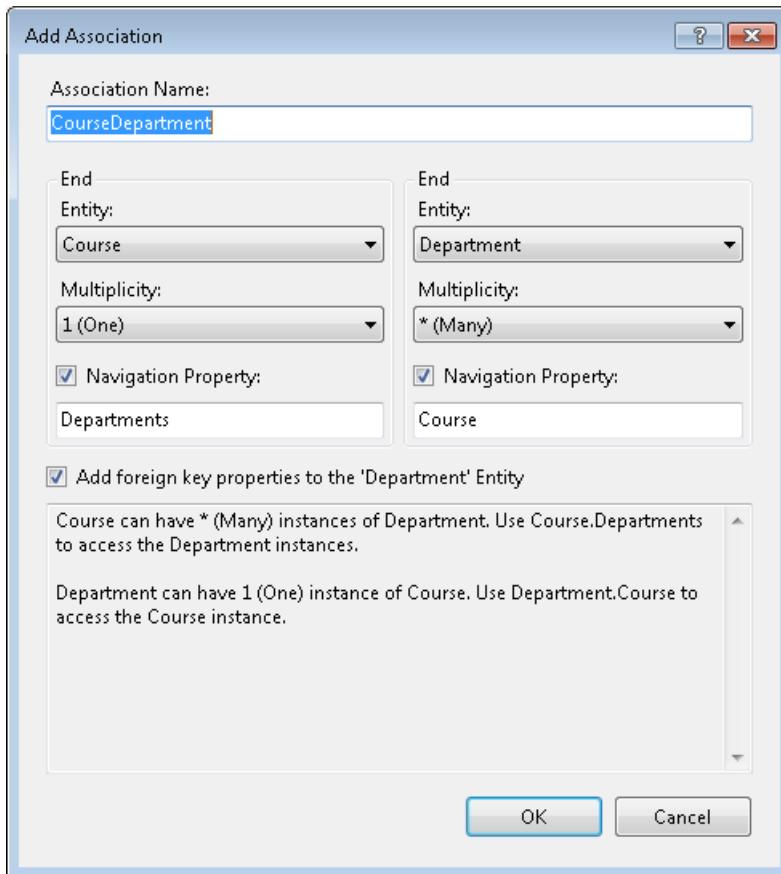
使用 EF 设计器创建关联会更新 .edmx 文件的模型内容。创建关联后，必须创建关联的映射(本主题后面将对此进行讨论)。

NOTE

本部分假定已添加要在模型之间创建关联的实体。

创建关联

- 右键单击设计图面的空白区域，指向“添加新项”，然后选择“关联...”。
- 在“添加关联”对话框中填写关联的设置。



NOTE

您可以通过清除“**导航属性**”并“**将外键属性添加到<实体类型名称>实体**”复选框，选择不向关联端的实体添加导航属性或外键属性。如果只添加一个导航属性，则将只能在一个方向遍历关联。如果不添加导航属性，则必须选择添加外键属性才能访问位于关联各端的实体。

- 单击“确定”。

删除关联

若要删除关联，请执行以下操作之一：

- 右键单击 EF 设计器图面上的关联，然后选择“删除”。
- 或 -
- 选择一个或多个关联并按 Delete 键。

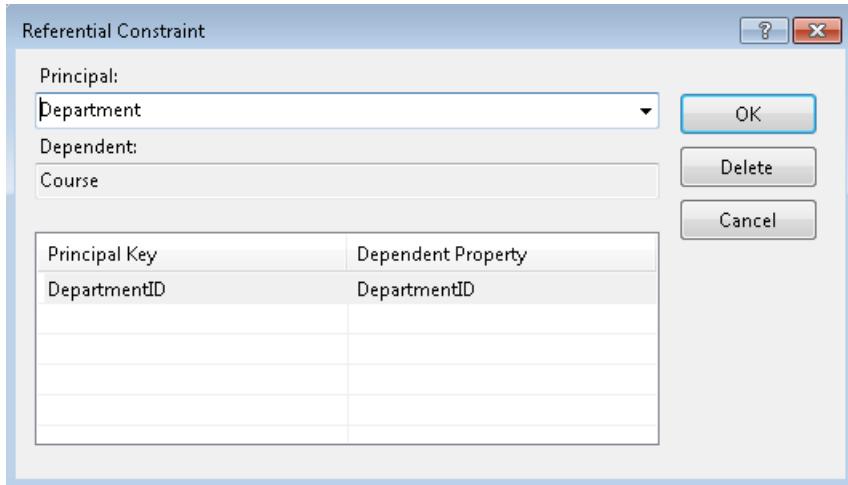
在实体中包括外键属性(引用约束)

建议你始终公开实体中关系的外键。实体框架使用引用约束来标识某个属性充当关系的外键。

如果在创建关系时选中了 "将外键属性添加到 <实体类型名称> 实体" 复选框，则会为你添加此引用约束。

使用 EF 设计器添加或编辑引用约束时，EF 设计器会在 .edmx 文件的 CSDL 内容中添加或修改 ReferentialConstraint 元素。

- 双击要编辑的关联。此时将显示 "引用约束" 对话框。
- 从 "主体" 下拉列表中，选择引用约束中的主体实体。实体的键属性将添加到对话框中的 "主体键" 列表中。
- 从 "依赖项" 下拉列表中，选择引用约束中的依赖实体。
- 对于具有依赖键的每个主体键，从 "依赖键" 列中的下拉列表中选择相应的依赖项。



- 单击 "确定"。

创建和编辑关联映射

可以在 EF 设计器的 "映射详细信息" 窗口中指定关联映射到数据库的方式。

NOTE

只能映射未指定引用约束的关联的详细信息。如果指定了引用约束，则会在实体中包含一个外键属性，您可以使用该实体的映射详细信息来控制外键映射到的列。

创建关联映射

- 右键单击设计图面中的关联，然后选择 "表映射"。这会在 "映射详细信息" 窗口中显示关联映射。
- 单击 "添加表或视图"。此时将显示一个下拉列表，其中包含存储模型中的所有表。
- 选择关联要映射到的表。"映射详细信息" 窗口将显示关联的两端以及每一端的实体类型的键属性。
- 对于每个键属性，请单击 "列" 字段，然后选择属性将映射到的列。

The screenshot shows the EntityDataSource Designer interface. On the left, the 'Course' entity is displayed with properties like CourseID, Title, Credits, and DepartmentID. On the right, the 'Person' entity is shown with properties such as PersonID, LastName, FirstName, HireDate, EnrollmentDate, and Discriminator. A bidirectional association named 'CourseInstructor' connects the two entities. Below the entities, the 'Mapping Details - CourseInstructor' window is open, showing the mapping between the CourseID and PersonID columns. The 'Property' column lists 'CourseID : Int32' and 'PersonID : Int32'. The 'Column' column lists 'CourseID : int' and 'PersonID : int'. The 'Operator' column shows a double-headed arrow indicating a bidirectional relationship.

编辑关联映射

- 右键单击设计图面上的关联，然后选择“表映射”。这会在“映射详细信息”窗口中显示关联映射。
- 单击“映射”<表名称>。此时将显示一个下拉列表，其中包含存储模型中的所有表。
- 选择关联要映射到的表。“映射详细信息”窗口将显示关联的两端以及每一端的实体类型的键属性。
- 对于每个键属性，请单击“列”字段，然后选择属性将映射到的列。

编辑和删除导航属性

导航属性是快捷方式属性，用于在模型中关联的两端查找实体。在创建两个实体类型之间的关联时可以创建导航属性。

编辑导航属性

- 选择 EF 设计器图面上的导航属性。有关导航属性的信息将显示在 Visual Studio 的“属性”窗口中。
- 在“属性”窗口中更改属性设置。

删除导航属性

- 如果概念模型中的实体类型上没有公开外键，则删除导航属性可能导致仅在一个方向遍历相应的关联，或者根本不遍历关联。
- 右键单击 EF 设计器图面上的导航属性，然后选择“删除”。

每个模型多个关系图

2020/3/12 •

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

此视频和页面显示了如何使用 Entity Framework Designer (EF 设计器) 将模型拆分为多个关系图。当模型变得太大，无法查看或编辑时，可能需要使用此功能。

在 EF 设计器的早期版本中，每个 EDMX 文件只能有一个关系图。从 Visual Studio 2012 开始，可以使用 EF 设计器将 EDMX 文件拆分成多个关系图。

观看视频

此视频演示如何使用 Entity Framework Designer (EF 设计器) 将模型拆分为多个关系图。当模型变得太大，无法查看或编辑时，可能需要使用此功能。

主讲人 :Julia Kornich

视频:WMV | [MP4](#) | [WMV \(ZIP\)](#)

EF 设计器概述

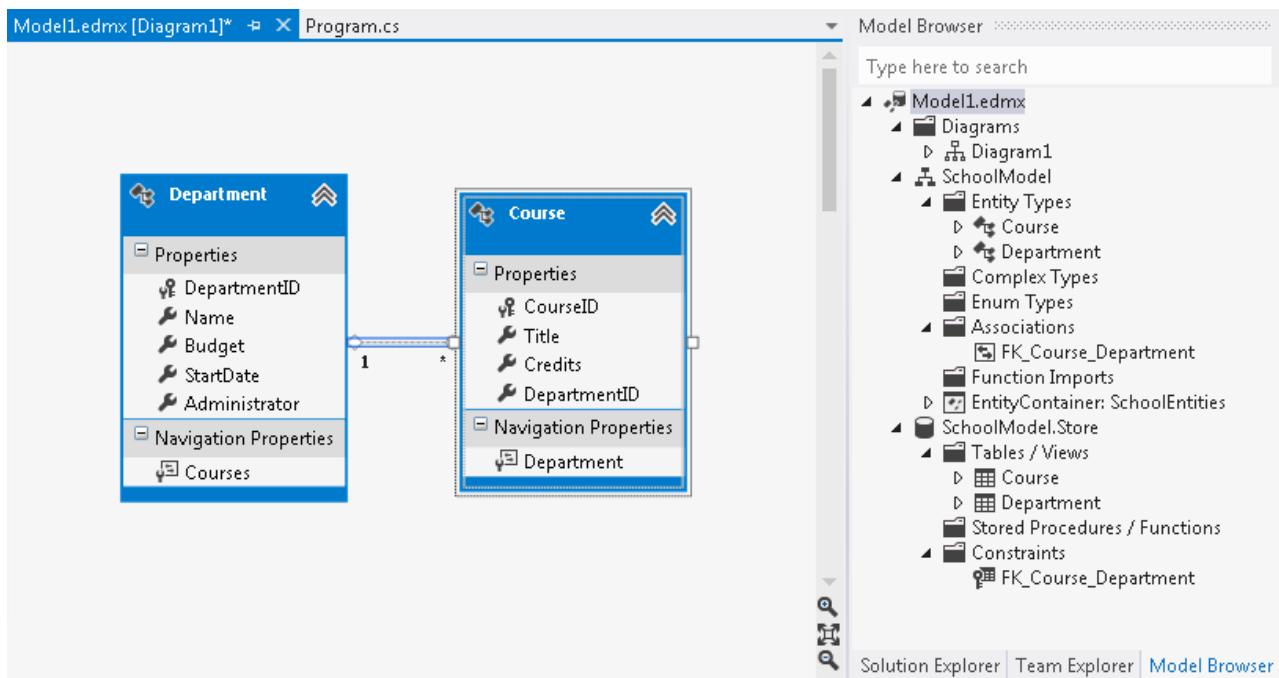
使用 EF 设计器的实体数据模型向导创建模型时，将创建一个 .edmx 文件并将其添加到解决方案中。此文件定义了你的实体的形状，以及它们如何映射到数据库。

EF 设计器包含以下组件：

- 用于编辑模型的可视化设计图面。您可以创建、修改或删除实体和关联。
- 模型浏览器 窗口提供模型的树视图。实体及其关联位于 *[ModelName]* 文件夹下。数据库表和约束位于 *[ModelName]* 下。存储文件夹。
- 用于查看和编辑映射的“映射详细信息”窗口。您可以将实体类型或关联映射到数据库表、列和存储过程。

实体数据模型向导完成时，将自动打开“可视化设计图面”窗口。如果“模型浏览器”不可见，请右键单击主设计图面，然后选择“模型浏览器”。

以下屏幕截图显示了在 EF 设计器中打开的 .edmx 文件。屏幕截图显示可视化设计图面(左侧)和 模型浏览器 窗口(右侧)。



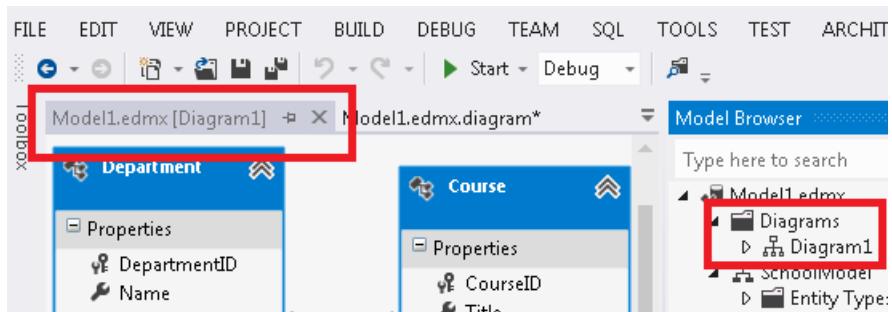
若要撤消在 EF 设计器中完成的操作, 请单击 "Ctrl + Z"。

使用关系图

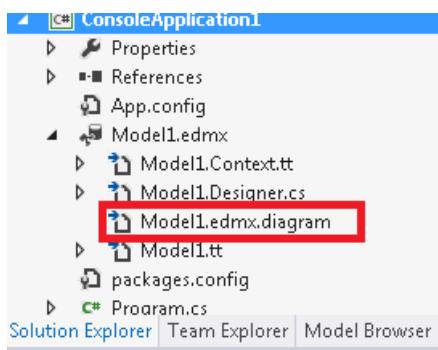
默认情况下, EF 设计器会创建一个名为 Diagram1 的关系图。如果你有一个具有大量实体和关联的关系图, 则你最喜欢按逻辑拆分它们。从 Visual Studio 2012 开始, 可以在多个关系图中查看概念模型。

添加新关系图时, 它们将显示在 "模型浏览器" 窗口中的 "关系图" 文件夹下。重命名关系图:在 "模型浏览器" 窗口中选择关系图, 单击 "名称", 然后键入新名称。您还可以右键单击关系图名称, 然后选择 "重命名"。

关系图名称将显示在 Visual Studio 编辑器中的 .edmx 文件名旁。例如, Model1[Diagram1]。



关系图内容(实体和关联的形状和颜色)存储在 .edmx 文件中。若要查看此文件, 请选择解决方案资源管理器并展开 .edmx 文件。



不应手动编辑 .edmx 文件, 此文件的内容可能会被 EF 设计器覆盖。

将实体和关联拆分为新关系图

您可以选择现有关系图上的实体(按住 Shift 键以选择多个实体)。单击鼠标右键并选择 "移到新关系图"。将创建新关系图，并将所选实体及其关联移动到关系图中。

或者，您可以在模型浏览器中右键单击 "关系图" 文件夹，然后选择 "添加新关系图"。然后，可以将实体从模型浏览器中的 "实体类型" 文件夹下拖放到设计图面上。

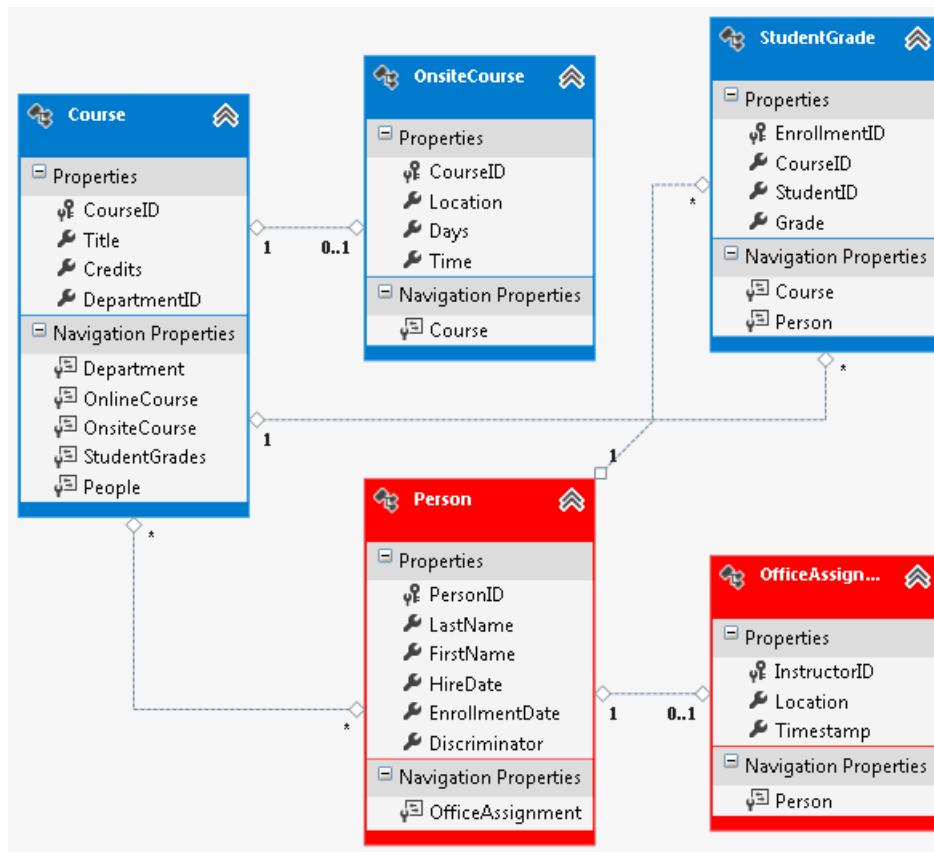
您还可以从一个关系图中剪切或复制实体(使用 Ctrl + X 或 Ctrl + C 键)，然后在另一个关系图中粘贴(使用 Ctrl + V 键)。如果要将实体粘贴到其中的关系图已经包含具有相同名称的实体，则将创建一个新实体并将其添加到模型中。例如：Diagram2 包含部门实体。然后，将其他部门粘贴到 Diagram2 上。将创建 Department1 实体并将其添加到概念模型中。

若要在关系图中包含相关实体，则右键单击该实体并选择 "包括相关项"。这将在指定的关系图中创建相关实体和关联的副本。

更改实体的颜色

除了将模型拆分为多个关系图外，还可以更改实体的颜色。

若要更改颜色，请在设计图面上选择一个或多个实体。然后，单击鼠标右键并选择 "属性"。在属性窗口中，选择 "填充颜色" 属性。使用有效的颜色名称(例如，Red)或有效的 RGB (例如，255、128、128)指定颜色。



Summary

在本主题中，我们介绍如何将模型拆分为多个关系图，以及如何使用 Entity Framework Designer 为实体指定不同的颜色。

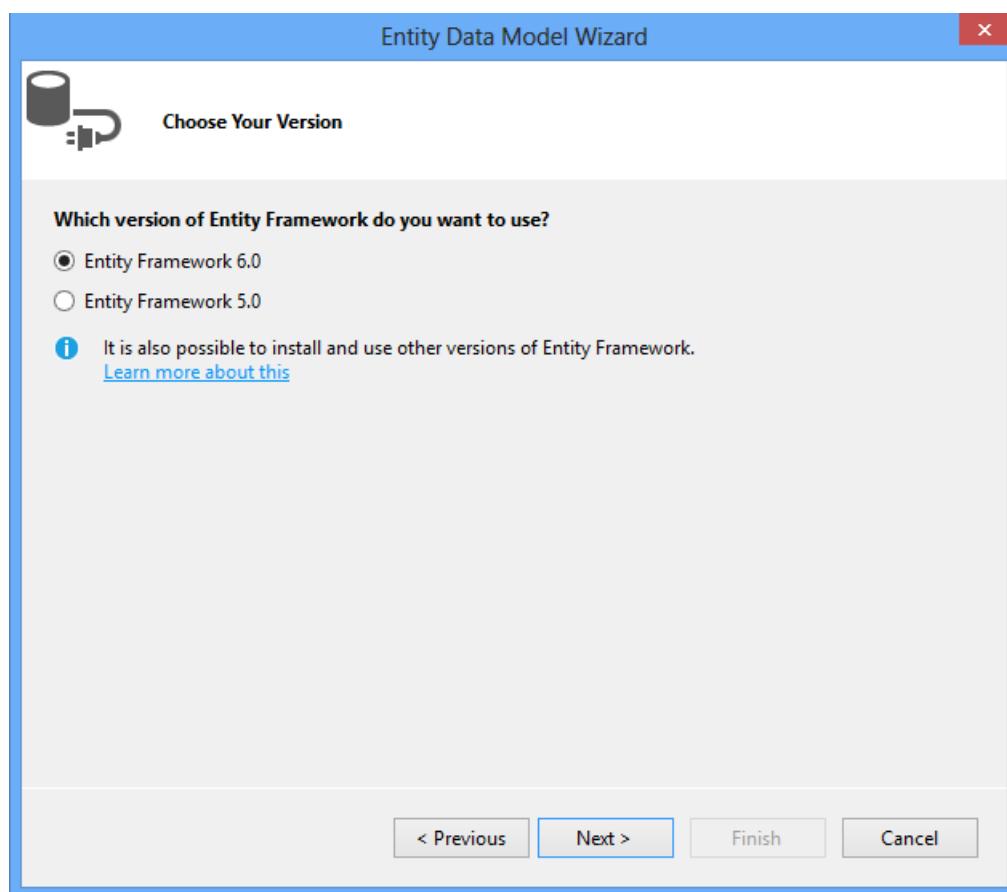
选择 EF 设计器模型实体框架运行时版本

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

从 EF6 开始，将以下屏幕添加到 EF 设计器，以便在创建模型时选择想要面向的运行时版本。当项目中尚未安装最新版本的实体框架时，将显示该屏幕。如果已安装最新版本，则默认情况下将使用它。



面向 EF6.x

你可以从“选择你的版本”屏幕中选择“EF6”，将 EF6 运行时添加到你的项目。添加 EF6 后，你将停止在当前项目中看到此屏幕。

如果已安装旧版 EF，则将禁用 EF6（因为不能将多个版本的运行时作为同一个项目的目标）。如果此处未启用 EF6 选项，请按照以下步骤将项目升级到 EF6：

1. 在解决方案资源管理器中右键单击项目，然后选择“管理 NuGet 包 ...”
2. 选择更新
3. 选择 EntityFramework (确保它将更新为所需的版本)
4. 单击“更新”。

面向 EF5.x

你可以从 "选择你的版本" 屏幕中选择 "EF5"，将 EF5 运行时添加到你的项目。添加 EF5 后，仍会看到此屏幕，其中 "EF6" 选项处于禁用状态。

如果已安装 EF4 版本的运行时，则会看到屏幕上列出的 EF 版本(而不是 EF5)。在这种情况下，可以使用以下步骤升级到 EF5：

1. 选择工具-> 库包管理器-> 程序包管理器控制台
2. 运行安装包 EntityFramework-版本 5.0.0

面向 EF4. x

可以使用以下步骤将 EF4 运行时安装到项目中：

1. 选择工具-> 库包管理器-> 程序包管理器控制台
2. 运行安装包 EntityFramework-版本 4.3.0

设计器代码生成模板

2020/4/8 ·

使用 Entity Framework 设计器创建模型时，会自动生成类和派生上下文。除默认代码生成之外，我们还提供了许多模板，可用于自定义生成的代码。这些模板以 T4 文本模板的形式提供，可按需自定义模板。

默认生成的代码取决于创建模型的 Visual Studio 版本：

- 在 Visual Studio 2012 和 2013 中创建的模型将生成简单的 POCO 实体类和派生的简化 DbContext 的上下文。
- 在 Visual Studio 2010 中创建的模型将生成派生自 EntityObject 的实体类和派生自 ObjectContext 的上下文。

NOTE

建议在添加模型后切换到 DbContext 生成器模板。

此页介绍了可用的模板，并说明了如何将模板添加到模型。

可用模板

以下模板由实体框架团队提供：

DbContext 生成器

此模板将生成简单的 POCO 实体类和派生自使用 EF6 的 DbContext 的上下文。除有需要使用以下列出的其他模板外，推荐使用此模板。如果使用的是近期版本的 Visual Studio(Studio 2013 及更高版本)，则它也是你默认获取的代码生成模板：创建新模型时，会默认使用此模板，并在你的 .edmx 文件下嵌套 T4 文件 (.tt)。

较旧版本的 Visual Studio

- Visual Studio 2012：要获取 EF 6.x DbContextGenerator 模板，需安装适用于 Visual Studio 的最新 Entity Framework Tools - 有关详细信息，请参阅[获取实体框架](#)页面。
- Visual Studio 2010：EF 6.x DbContextGenerator 模板不适用于 Visual Studio 2010。

EF 5.x 的 DbContext 生成器

如果使用的是较旧版本的 EntityFramework NuGet 包(具有主要版本 5)，则需使用 EF 5.x DbContext 生成器模板。

如果使用的是 Visual Studio 2013 或 2012，则其中已安装该模板。

如果使用的是 Visual Studio 2010，添加模板时，则需选择“联机”选项卡，从 Visual Studio 库中下载模板。或者可直接从 Visual Studio 库提前安装模板。因为模板包含在较新版本的 Visual Studio 中，因此只能将库上的版本安装在 Visual Studio 2010 中。

- [适用于 C# 的 EF 5.x DbContext 生成器](#)
- [适用于 C# 网站的 EF 5.x DbContext 生成器](#)
- [适用于 VB.NET 的 EF 5.x DbContext 生成器](#)
- [适用于 VB.NET 网站的 EF 5.x DbContext 生成器](#)

EF 4.x 的 DbContext 生成器

如果使用的是较旧版本的 EntityFramework NuGet 包(具有主要版本 4)，则需使用 EF 4.x DbContext 生成器模板。添加模板时，可在“联机”选项卡中找到此模板，或可直接从 Visual Studio 库提前安装模板。

- [适用于 C# 的 EF 4.x DbContext 生成器](#)
- [适用于 C# 网站的 EF 4.x DbContext 生成器](#)
- [适用于 VB.NET 的 EF 4.x DbContext 生成器](#)

- 适用于 VB.NET 网站的 EF 4.x DbContext 生成器

EntityObject 生成器

此模板将生成从 EntityObject 派生的实体类和从 ObjectContext 派生的上下文。

NOTE

请考虑使用 DbContext 生成器

现推荐新应用程序使用 DbContext 生成器模板。DbContext 生成器利用了更简单的 DbContext API。EntityObject 生成器仍支持用于现有应用程序。

Visual Studio 2010、2012 和 2013

添加模板时，需选择“联机”选项卡，从 Visual Studio 库中下载模板。或可直接从 Visual Studio 库提前安装模板。

- 适用于 C# 的 EF 6.x EntityObject 生成器
- 适用于 C# 网站的 EF 6.x EntityObject 生成器
- 适用于 VB.NET 的 EF 6.x EntityObject 生成器
- 适用于 VB.NET 网站的 EF 6.x EntityObject 生成器

EF 5.x 的 EntityObject 生成器

如果使用的是 Visual Studio 2012 或 2013，添加模板时，则需选择“联机”选项卡，从 Visual Studio 库中下载模板。或可直接从 Visual Studio 库提前安装模板。因为模板包含在 Visual Studio 2010 中，因此只能将库上的模板安装在 Visual Studio 2012 和 2013 中。

- 适用于 C# 的 EF 5.x EntityObject 生成器
- 适用于 C# 网站的 EF 5.x EntityObject 生成器
- 适用于 VB.NET 的 EF 5.x EntityObject 生成器
- 适用于 VB.NET 网站的 EF 5.x EntityObject 生成器

如果只需要生成 ObjectContext 代码，无需编辑模板，则可还原为 EntityObject 代码生成。

如果使用的是 Visual Studio 2010，则其中已安装该模板。如果在 Visual Studio 2010 中创建新模型，则默认使用此模板，但项目中不包含 .tt 文件。若想自定义模板，则需将其添加到项目中。

自跟踪实体 (STE) 生成器

此模板将生成自跟踪实体类和从 ObjectContext 派生的上下文。在 EF 应用程序中，上下文负责跟踪实体中的更改。但是，在 N 层方案中，上下文可能无法用于修改实体的层。自跟踪实体有助于跟踪任意层中的更改。有关详细信息，请参阅[自跟踪实体](#)。

NOTE

不推荐使用 STE 模板

不再建议在新应用程序中使用 STE 模板，虽然该模板仍支持用于现有应用程序。请查看[断开连接的实体文章](#)了解推荐用于 N 层方案的其他选项。

NOTE

没有 EF 6.x 版的 STE 模板。

NOTE

没有 Visual Studio 2013 版的 STE 模板。

Visual Studio 2012

如果使用的是 Visual Studio 2012，添加模板时，则需选择“联机”选项卡，从 Visual Studio 库中下载模板。或可直接从 Visual Studio 库提前安装模板。因为模板包含在 Visual Studio 2010 中，因此只能将库上的模板安装在 Visual Studio 2012 中。

- [适用于 C# 的 EF 5.x STE 生成器](#)
- [适用于 C# 网站的 EF 5.x STE 生成器](#)
- [适用于 VB.NET 的 EF 5.x STE 生成器](#)
- [适用于 VB.NET 网站的 EF 5.x STE 生成器](#)

Visual Studio 2010**

如果使用的是 Visual Studio 2010，则其中已安装该模板。

POCO 实体生成器

此模板将生成 POCO 实体类和从 ObjectContext 派生的上下文

NOTE

请考虑使用 DbContext 生成器

现推荐使用 DbContext 生成器模板在新应用程序中生成 POCO 类。DbContext 生成器利用了新的 DbContext API，且可生成更简单的 POCO 类。POCO 实体生成器仍支持用于现有应用程序。

NOTE

没有 EF 5.x 或 EF 6.x 版的 STE 模板。

NOTE

没有 Visual Studio 2013 版的 POCO 模板。

Visual Studio 2012 和 Visual Studio 2010

添加模板时，需选择“联机”选项卡，从 Visual Studio 库中下载模板。或可直接从 Visual Studio 库提前安装模板。

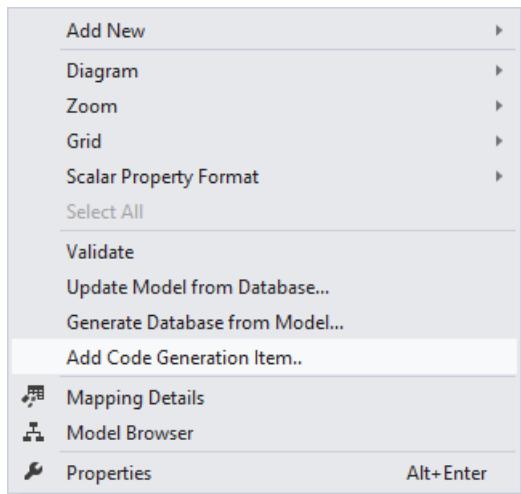
- [适用于 C# 的 EF 4.x POCO 生成器](#)
- [适用于 C# 网站的 EF 4.x POCO 生成器](#)
- [适用于 VB.NET 的 EF 4.x POCO 生成器](#)
- [适用于 VB.NET 网站的 EF 4.x POCO 生成器](#)

“网站”模板是什么

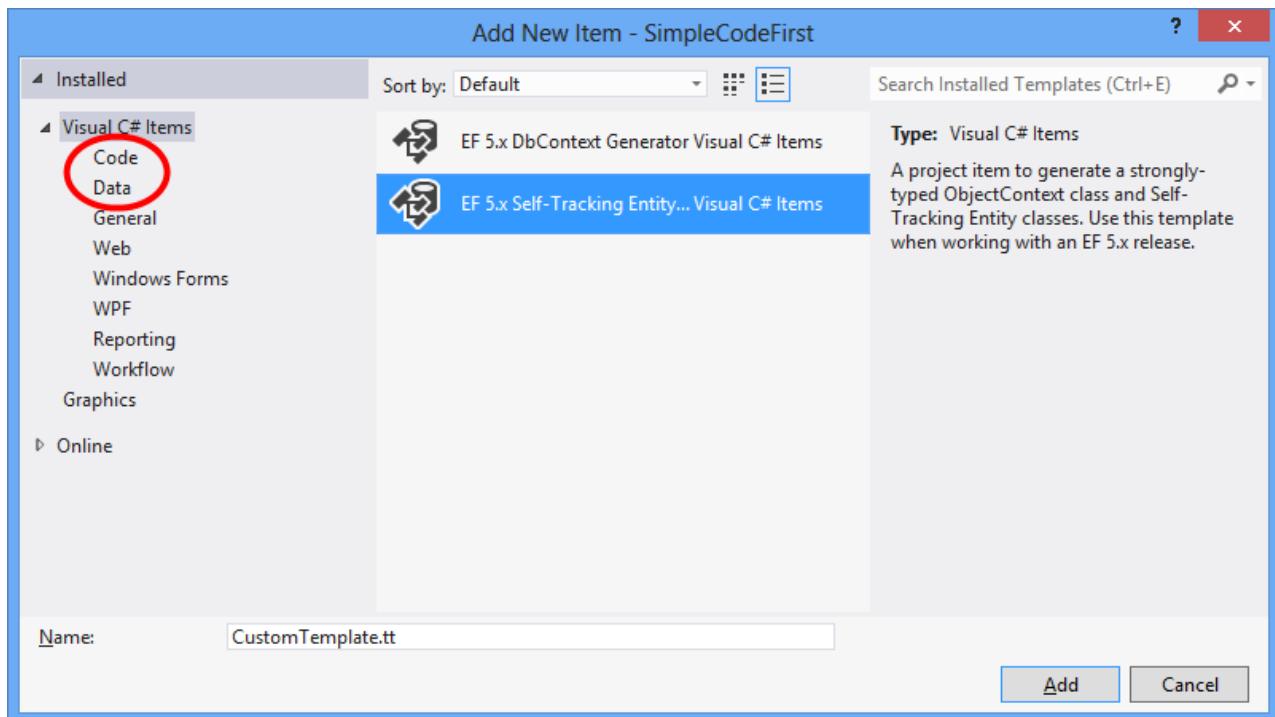
“网站”模板（例如适用于 C# 网站的 EF 5.x DbContext 生成器）可用于通过“文件”->“新建”->“网站...”创建的网站项目。这不同于通过“文件”->“新建”->“项目...”创建的 Web 应用程序，后者使用的是标准模板。我们单独提供了这些模板，因为 Visual Studio 中的项模板系统需要这些模板。

使用模板

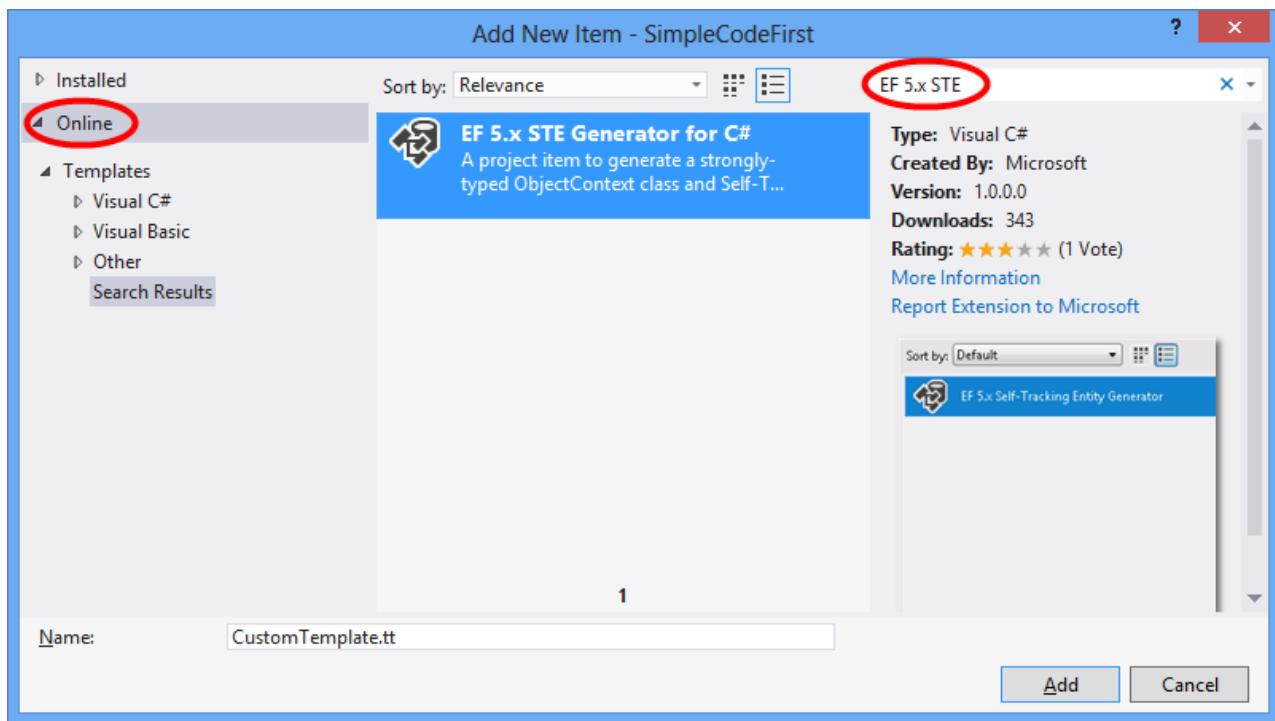
若要开始使用代码生成模板，请右键单击 EF 设计器中设计界面上的空白区域，然后选择“添加代码生成项...”。



如果已安装要使用的模板(或模板包含在 Visual Studio 中), 则可从左侧菜单的“代码”或“数据”部分使用此模板。



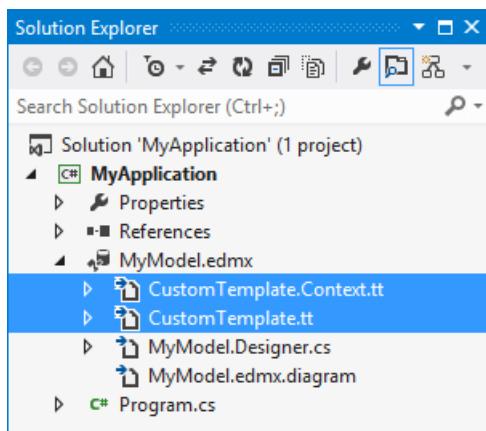
如果尚未安装模板, 请选择左侧菜单中的“联机”并搜索所需模板。



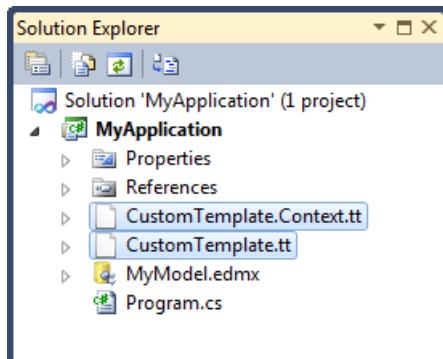
如果使用的是 Visual Studio 2012，则新的 .tt 文件将嵌套在 .edmx 文件下。*

NOTE

对于在 Visual Studio 2012 中创建的模型，需删除用于默认代码生成的模板，否则会生成重复的类和上下文。默认文件是 <model name>.tt 和 <model name>.context.tt。



如果使用的是 Visual Studio 2010，则 tt 文件会直接添加到项目中。



恢复到 Entity Framework Designer 中的 ObjectContext

2020/3/11 •

使用早期版本的实体框架使用 EF 设计器创建的模型将生成派生自 EntityObject 的 ObjectContext 和实体类的上下文。

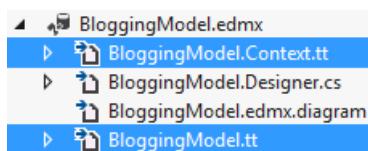
从 EF 4.1 开始，我们建议换到代码生成模板，该模板生成派生自 DbContext 和 POCO 实体类的上下文。

在 Visual Studio 2012 中，会为使用 EF 设计器创建的所有新模型获取默认生成的 DbContext 代码。现有模型将继续生成基于 ObjectContext 的代码，除非你决定要切换到基于 DbContext 的代码生成器。

恢复到 ObjectContext 代码生成

1. 禁用 DbContext 代码生成

派生的 DbContext 和 POCO 类的生成由项目中的两个 tt 文件处理，如果在 "解决方案资源管理器" 中展开 .edmx 文件，则会看到这些文件。从项目中删除这两个文件。



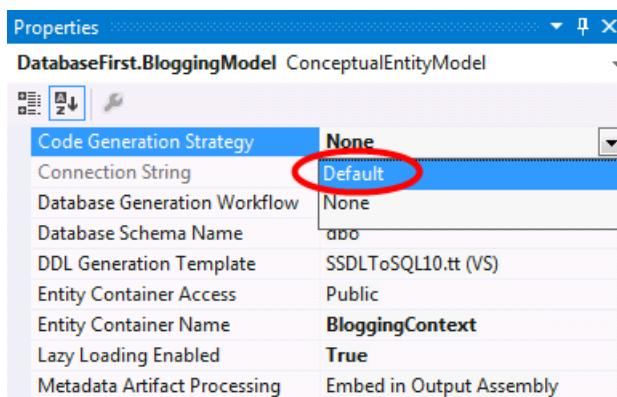
如果你使用的是 VB.NET，则需要选择“显示所有文件”按钮以查看嵌套的文件。



2. 重新启用 ObjectContext 代码生成

在 EF 设计器中打开模型，右键单击设计图面的空白部分，然后选择“属性”。

在属性窗口将代码生成策略从None更改为默认值。



CSDL 规范

2020/3/11 ·

概念架构定义语言 (CSDL) 是一种基于 XML 的语言，它描述构成数据驱动应用程序的概念模型的实体、关系和函数。此概念模型可由实体框架或 WCF 数据服务使用。实体框架使用 CSDL 描述的元数据将概念模型中定义的实体和关系映射到数据源。有关详细信息，请参阅[SSDL 规范](#)和[MSL 规范](#)。

CSDL 是实体数据模型的实体框架实现。

在实体框架应用程序中，概念模型元数据从 csdl 文件(用 CSDL 编写)加载到 EdmItemCollection 的实例中，可以使用中的方法进行访问。“System.object”类。实体框架使用概念模型元数据将针对概念模型的查询转换为特定于数据源的命令。

EF 设计器会在设计时将概念模型信息存储在 .edmx 文件中。在生成时，EF 设计器使用 .edmx 文件中的信息来创建运行时实体框架所需的 csdl 文件。

CSDL 的版本按 XML 命名空间进行区分。

CSDL 版本	XML 命名空间
CSDL v1	https://schemas.microsoft.com/ado/2006/04/edm
CSDL v2	https://schemas.microsoft.com/ado/2008/09/edm
CSDL v3	https://schemas.microsoft.com/ado/2009/11/edm

Association 元素 (CSDL)

Association元素定义了两个实体类型之间的关系。关联必须指定关系中涉及的实体类型和关系的每一端可能的实体类型数量(也称为重数)。关联端的多重性值可以为一(1)、零或一(0 ..1)或多(*)。此信息在两个 End 子元素中指定。

通过导航属性或外键(如果在实体类型上公开了外键)可以访问关联一端的实体类型实例。

在应用程序中，关联的实例表示实体类型的实例之间的特定关联。关联实例按逻辑分组在关联集中。

Association元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- End (正好2个元素)
- ReferentialConstraint (零个或一个元素)
- 批注元素(零个或多个元素)

适用的属性

下表描述了可应用于Association元素的特性。

特性	说明	示例
■	是	关联的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Association元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个association元素, 该元素定义在Customer和Order实体类型上没有公开外键时的CustomerOrders关联。关联的每个端点的多重性值指示可以将多个订单与一个客户关联, 但只能有一个客户与一个订单相关联。此外, OnDelete元素指示在删除客户时, 将删除与特定客户相关且已加载到ObjectContext的所有订单。

```
<Association Name="CustomerOrders">
    <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" >
        <OnDelete Action="Cascade" />
    </End>
    <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

下面的示例演示一个association元素, 该元素定义在Customer和Order实体类型上公开外键时的CustomerOrders关联。公开外键后, 实体之间的关系将通过ReferentialConstraint元素进行管理。将此关联映射到数据源并不需要 AssociationSetMapping 元素。

```
<Association Name="CustomerOrders">
    <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" >
        <OnDelete Action="Cascade" />
    </End>
    <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
    <ReferentialConstraint>
        <Principal Role="Customer">
            <PropertyRef Name="Id" />
        </Principal>
        <Dependent Role="Order">
            <PropertyRef Name="CustomerId" />
        </Dependent>
    </ReferentialConstraint>
</Association>
```

AssociationSet 元素 (CSDL)

概念架构定义语言(CSDL)中的AssociationSet元素是同一类型的关联实例的逻辑容器。关联集为对关联实例进行分组提供了定义, 以便能够将它们映射到数据源。

AssociationSet元素可以具有以下子元素(按所列顺序):

- 文档(允许零个或一个元素)
- End (只需要两个元素)
- Annotation 元素(允许零个或多个元素)

Association特性指定关联集包含的关联类型。构成关联集末尾的实体集是通过恰好两个子End元素指定的。

适用的属性

下表介绍可应用于AssociationSet元素的特性。

属性	说明	示例
■ Name	是	实体集的名称。Name特性的值不能与Association特性的值相同。
■ Association	是	关联集包含其实例的关联的完全限定名称。关联必须与关联集位于同一个命名空间中。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于AssociationSet元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有两个AssociationSet元素的EntityContainer元素：

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

CollectionType 元素 (CSDL)

概念架构定义语言(CSDL)中的CollectionType元素指定函数参数或函数返回类型是集合。CollectionType元素可以是参数元素的子元素, 也可以是 ReturnType (函数)元素的子元素。可以使用type特性或以下子元素之一指定集合类型：

- CollectionType
- ReferenceType
- RowType
- TypeRef

NOTE

如果集合类型是使用type特性和子元素指定的，则模型不会进行验证。

适用的属性

下表介绍可应用于CollectionType元素的特性。请注意，DefaultValue、MaxLength、FixedLength、精度、小数位数、Unicode和排序规则特性仅适用于edmsimpletype的集合。

属性	说明	示例
类型	是	集合的类型。
■ Null	是	True(默认值)或False, 取决于属性是否可以具有null值。 [!NOTE]
> 在 CSDL v1 中, 复杂类型属性必须具有 <code>Nullable="False"</code> 。		
■	是	属性的默认值。
MaxLength	是	属性值的最大长度。
FixedLength	是	True或False, 具体取决于属性值是否将作为固定长度字符串存储。
■	是	属性值的精度。
■	是	属性值的刻度。
SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息, 请参阅 SRID and SRID (SQL Server)
Unicode	是	True或False, 具体取决于属性值是否将存储为 Unicode 字符串。
■	是	指定要在数据源中使用的排序的字符串。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于CollectionType元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个模型定义函数, 该函数使用CollectionType元素来指定该函数返回Person实体类型的集合(使用ElementType特性指定)。

```

<Function Name="LastNamesAfter">
    <Parameter Name="someString" Type="Edm.String"/>
    <ReturnType>
        <CollectionType ElementType="SchoolModel.Person"/>
    </ReturnType>
    <DefiningExpression>
        SELECT VALUE p
        FROM SchoolEntities.People AS p
        WHERE p.LastName >= someString
    </DefiningExpression>
</Function>

```

下面的示例演示一个模型定义函数，该函数使用`CollectionType`元素来指定该函数返回行的集合（在`RowType`元素中指定）。

```

<Function Name="LastNamesAfter">
    <Parameter Name="someString" Type="Edm.String" />
    <ReturnType>
        <CollectionType>
            <RowType>
                <Property Name="FirstName" Type="Edm.String" Nullable="false" />
                <Property Name="LastName" Type="Edm.String" Nullable="false" />
            </RowType>
        </CollectionType>
    </ReturnType>
    <DefiningExpression>
        SELECT VALUE ROW(p.FirstName, p.LastName)
        FROM SchoolEntities.People AS p
        WHERE p.LastName >= somestring
    </DefiningExpression>
</Function>

```

下面的示例演示一个模型定义的函数，该函数使用`CollectionType`元素来指定该函数接受作为参数的部门实体类型的集合。

```

<Function Name="GetAvgBudget">
    <Parameter Name="Departments">
        <CollectionType>
            <TypeRef Type="SchoolModel.Department"/>
        </CollectionType>
    </Parameter>
    <ReturnType Type="Collection(Edm.Decimal)"/>
    <DefiningExpression>
        SELECT VALUE AVG(d.Budget) FROM Departments AS d
    </DefiningExpression>
</Function>

```

ComplexType 元素 (CSDL)

`ComplexType`元素定义由`EdmSimpleType`属性或其他复杂类型组成的数据结构。复杂类型可以是实体类型或其他复杂类型的属性。复杂类型类似于复杂类型定义数据中的实体类型。但是，在复杂类型与实体类型之间仍存在着一些重要区别：

- 复杂类型没有标识(或键)，因此不能独立存在。复杂类型只能作为实体类型或其他复杂类型的属性而存在。
- 复杂类型不能参与关联。关联的任一端都不能是复杂类型，因此不能为复杂类型定义导航属性。
- 复杂类型属性不能具有 null 值，但可以将复杂类型的每个标量属性设置为 null。

ComplexType元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- Property(零个或多个元素)
- 批注元素(零个或多个元素)

下表介绍可应用于**ComplexType**元素的特性。

名称	是否	说明
名称	是	复杂类型的名称。复杂类型的名称不能与模型作用域中的其他复杂类型、实体类型或关联的名称相同。
BaseType	是	作为所定义的复杂类型的基类型的另一个复杂类型的名称。 [!NOTE]
> 此属性在 CSDL v1 中不适用。在该版本中不支持复杂类型的继承。		
摘要	是	True或False (默认值)，取决于复杂类型是否为抽象类型。 [!NOTE]
> 此属性在 CSDL v1 中不适用。该版本中的复杂类型不能是抽象类型。		

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**ComplexType**元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例显示了一个复杂类型Address，其中包含EdmSimpleType属性StreetAddress、City、StateOrProvince、国家/地区和邮政编码。

```
<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>
```

若要将复杂类型地址(如上所示)定义为某个实体类型的属性，必须在实体类型定义中声明该属性类型。下面的示

例显示了在实体类型(发布者)上作为复杂类型的地址属性:

```
<EntityType Name="Publisher">
  <Key>
    <PropertyRef Name="Id" />
  </Key>
  <Property Type="Int32" Name="Id" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false" />
  <Property Type="BooksModel.Address" Name="Address" Nullable="false" />
  <NavigationProperty Name="Books" Relationship="BooksModel.PublishedBy"
    FromRole="Publisher" ToRole="Book" />
</EntityType>
```

DefiningExpression 元素 (CSDL)

概念架构定义语言(CSDL)中的DefiningExpression元素包含一个定义概念模型中的函数的实体 SQL 表达式。

NOTE

出于验证目的, DefiningExpression元素可以包含任意内容。但是, 如果DefiningExpression元素不包含有效的实体 SQL, 则实体框架在运行时将引发异常。

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于DefiningExpression元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例使用DefiningExpression元素来定义一个函数, 该函数返回发布书籍之后的年数。

DefiningExpression元素的内容是以实体 SQL 编写的。

```
<Function Name="GetYearsInPrint" ReturnType="Edm.Int32" >
  <Parameter Name="book" Type="BooksModel.Book" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
  </DefiningExpression>
</Function>
```

Dependent 元素 (CSDL)

概念架构定义语言(CSDL)中的依赖元素是 ReferentialConstraint 元素的子元素, 用于定义引用约束的依赖端。

ReferentialConstraint元素定义的功能与关系数据库中的引用完整性约束类似。与数据库表中的一个(或多个)列可以引用另一个表的主键相同, 实体类型的一个(或多个)属性可以引用另一个实体类型的实体键。引用的实体类型称为约束的主体端。引用主体端的实体类型称为约束的依赖端。PropertyRef元素用于指定哪些键引用主体端。

依赖元素可以具有以下子元素(按所列顺序):

- PropertyRef (一个或多个元素)

- 批注元素(零个或多个元素)

适用的属性

下表描述了可应用于依赖元素的特性。

■	■	■
■	是	关联的依赖端的实体类型的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了在PublishedBy关联的定义中使用的ReferentialConstraint元素。Book实体类型的PublisherId属性构成了引用约束的依赖端。

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Documentation 元素 (CSDL)

概念架构定义语言(CSDL)中的文档元素可用于提供有关在父元素中定义的对象的信息。在.edmx 文件中, 当文档元素是元素的子元素, 而该元素显示为 EF 设计器设计图面上的对象(如实体、关联或属性)时, 文档元素的内容将显示在对象的 Visual Studio 属性窗口中。

文档元素可以具有以下子元素(按所列顺序):

- 摘要:父元素的简要说明。(零个或一个元素)
- LongDescription:父元素的详细说明。(零个或一个元素)
- 批注元素.(零个或多个元素)

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于文档元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例将文档元素显示为 EntityType 元素的子元素。如果下面的代码片段在 .edmx 文件的 CSDL 内容中，则当你单击 Customer 实体类型时，Summary 和 LongDescription 元素的内容将显示在 Visual Studio 的“属性”窗口中。

```
<EntityType Name="Customer">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Type="Int32" Name="CustomerId" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false" />
</EntityType>
```

End 元素 (CSDL)

概念架构定义语言 (CSDL) 中的 End 元素可以是 Association 元素或 AssociationSet 元素的子元素。在每种情况下，End 元素的角色不同，适用的属性也不同。

End 元素作为 Association 元素的子元素

结束元素（作为 association 元素的子元素）标识了关联一端的实体类型以及该关联端可以存在的实体类型实例的数量。关联端定义为关联的一部分；关联必须正好有两个关联端。通过导航属性或外键（如果实体类型上有）可以访问关联某一端的实体类型实例。

结束元素可以具有以下子元素（按所列顺序）：

- 文档（零个或一个元素）
- OnDelete（零个或一个元素）
- 批注元素（零个或多个元素）

适用的属性

下表描述了当结束元素是 Association 元素的子元素时，可应用于该元素的特性。

属性	说明	示例
类型	是	关联一端的实体类型的名称。
■	是	关联端的名称。如果不提供名称，将使用关联端的实体类型的名称。
■	是	1、0 1 或 *，具体取决于可在关联末尾的实体类型实例的数量。 1 指示关联端刚好存在一个实体类型实例。 0..0 表示在关联端存在零个或一个实体类型实例。 * 指示关联端存在零个、一个或多个实体类型实例。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个定义CustomerOrders关联的association元素。关联的每个端点的多重性值指示可以将多个订单与一个客户关联, 但只能有一个客户与一个订单相关联。此外, OnDelete元素指示在删除客户时, 将删除与特定客户相关且已加载到 ObjectContext 的所有订单。

```
<Association Name="CustomerOrders">
  <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1" />
  <End Type="ExampleModel.Order" Role="Order" Multiplicity="*">
    <OnDelete Action="Cascade" />
  </End>
</Association>
```

End 元素作为 AssociationSet 元素的子元素

End元素指定关联集的一端。AssociationSet元素必须包含两个结束元素。End元素中包含的信息用于将关联集映射到数据源。

结束元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 批注元素(零个或多个元素)

NOTE

Annotation 元素必须出现在所有其他子元素之后。仅在 CSDL v2 和更高版本中允许使用批注元素。

适用的属性

下表描述了当结束元素为AssociationSet元素的子元素时, 可应用于该元素的特性。

■	■	■
EntitySet	是	定义父AssociationSet元素一端的EntitySet元素的名称。必须在与父AssociationSet元素相同的实体容器中定义EntitySet元素。
■	是	关联集端的名称。如果未使用Role属性, 则关联集端的名称将为实体集的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有两个AssociationSet元素的EntityContainer元素, 每个元素都有两个End元素:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

EntityContainer 元素 (CSDL)

概念架构定义语言(CSDL)中的EntityContainer元素是实体集、关联集和函数导入的逻辑容器。概念模型实体容器通过 EntityContainerMapping 元素映射到存储模型实体容器。存储模型实体容器描述数据库的结构:实体集描述表、关联集描述外键约束、函数导入描述数据库中的存储过程。

EntityContainer元素可以有零个或一个文档元素。如果存在文档元素, 则它必须位于所有EntitySet、AssociationSet和FunctionImport元素之前。

EntityContainer元素可包含零个或多个下列子元素(按所列顺序):

- EntitySet
- AssociationSet
- FunctionImport
- 批注元素

可以扩展EntityContainer元素, 使其包含同一命名空间中的另一个entitycontainer的内容。若要包含另一个entitycontainer的内容, 请在引用entitycontainer元素中, 将 "扩展" 属性的值设置为要包括的EntityContainer元素的名称。包含的entitycontainer元素的所有子元素将被视为引用entitycontainer元素的子元素。

适用的属性

下表描述了可应用于Using元素的特性。

■	■	■
■	是	实体容器的名称。

■	是	同一命名空间中另一实体容器的名称。
---	---	-------------------

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于EntityContainer元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个EntityContainer元素, 该元素定义三个实体集和两个关联集。

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

EntitySet 元素 (CSDL)

概念架构定义语言中的EntitySet元素是某个实体类型的实例和从该实体类型派生的任何类型的实例的逻辑容器。实体类型与实体集之间的关系类似于关系数据库中行与表之间的关系。实体类型与行类似, 它定义一组相关数据; 而实体集与表类似, 它包含该定义的实例。实体集为对实体类型实例分组提供了一个构造, 以便能够将它们映射到数据源中的相关数据结构。

可以为特定实体类型定义多个实体集。

NOTE

EF 设计器不支持每个类型包含多个实体集的概念模型。

EntitySet元素可以具有以下子元素(按所列顺序) :

- 文档元素(允许零个或一个元素)
- Annotation 元素(允许零个或多个元素)

适用的属性

下表描述了可应用于EntitySet元素的特性。

属性	说明	示例
■	是	实体集的名称。
EntityType	是	实体集包含其实例的实体类型的完全限定名称。

NOTE

可以向**EntitySet**元素应用任意数量的批注特性(自定义 XML 特性)。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有三个**EntitySet**元素的**EntityContainer**元素:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="WrittenBy" Association="BooksModel.WrittenBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

每个类型可以定义多个实体集 (MEST)。下面的示例定义了一个实体容器, 其中包含书籍实体类型的两个实体集:

```
<EntityContainer Name="BooksContainer" >
  <EntitySet Name="Books" EntityType="BooksModel.Book" />
  <EntitySet Name="FictionBooks" EntityType="BooksModel.Book" />
  <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
  <EntitySet Name="Authors" EntityType="BooksModel.Author" />
  <AssociationSet Name="PublishedBy" Association="BooksModel.PublishedBy">
    <End Role="Book" EntitySet="Books" />
    <End Role="Publisher" EntitySet="Publishers" />
  </AssociationSet>
  <AssociationSet Name="BookAuthor" Association="BooksModel.BookAuthor">
    <End Role="Book" EntitySet="Books" />
    <End Role="Author" EntitySet="Authors" />
  </AssociationSet>
</EntityContainer>
```

EntityType 元素 (CSDL)

EntityType元素表示概念模型中的顶级概念(例如客户或订单)的结构。实体类型是应用程序中实体类型实例的模板。每个模板都包含以下信息：

- 唯一名称。(必选。)
- 由一个或多个属性定义的实体键。(必选。)
- 用于包含数据的属性。(可选。)
- 导航属性, 用于从关联的一端导航至另一端。(可选。)

在应用程序中, 实体类型的实例表示一个特定对象(例如特定客户或订单)。实体类型的每个实例在某个实体集中都必须具有一个唯一的实体键。

只有两个实体类型实例的类型相同且其实体键的值也相同时, 才认为它们是相等的。

EntityType元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- Key(零个或一个元素)
- Property(零个或多个元素)
- NavigationProperty(零个或多个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于**EntityType**元素的特性。

属性	说明	示例
■	是	实体类型的名称。
BaseType	是	作为正在定义的实体类型的基类型的另一个实体类型。
■	是	True或False, 具体取决于实体类型是否为抽象类型。
OpenType	是	True或False, 具体取决于实体类型是否为开放式实体类型。 [!NOTE]
> OpenType 特性仅适用于在与 ADO.NET Data Services 一起使用的概念 模型中定义的实体类型。		

NOTE

可以向**EntityType**元素应用任意数量的批注特性(自定义 XML 特性)。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有三个属性元素和两个NavigationProperty元素的**EntityType**元素：

```

<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>

```

EnumType 元素 (CSDL)

EnumType元素表示一个枚举类型。

EnumType元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 成员(零个或多个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于EnumType元素的特性。

属性	说明	示例
Name	是	实体类型的名称。
IsFlags	是	True或False, 具体取决于枚举类型是否可用作一组标志。默认值为False。
UnderlyingType	是	Edm、edm、edm、edm或edm，定义类型的值的范围的。 枚举元素的默认基础类型为Edm。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于EnumType元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有三个成员元素的EnumType元素：

```

<EnumType Name="Color" IsFlags="false" UnderlyingTyp="Edm.Byte">
  <Member Name="Red" />
  <Member Name="Green" />
  <Member Name="Blue" />
</EntityType>

```

Function 元素 (CSDL)

概念架构定义语言(CSDL)中的函数元素用于在概念模型中定义或声明函数。函数通过使用 DefiningExpression 元素来定义。

函数元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- Parameter(零个或多个元素)
- DefiningExpression(零个或一个元素)
- ReturnType (函数)(零个或一个元素)
- 批注元素(零个或多个元素)

函数的返回类型必须与 `returntype` (function) 元素或 `returntype` 属性(见下文)一起指定, 但不能同时指定两者。可能的返回类型为任何 EdmSimpleType、实体类型、复杂类型、行类型或引用类型(或这些类型之一的集合)。

适用的属性

下表描述了可应用于 Function 元素的特性。

属性	必需	说明
■	是	函数的名称。
ReturnType	是	函数返回的类型。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于 Function 元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例使用 function 元素来定义一个函数, 该函数返回一个指导员后的年数。

```

<Function Name="YearsSince" ReturnType="Edm.Int32">
  <Parameter Name="date" Type="Edm.DateTime" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(date)
  </DefiningExpression>
</Function>

```

FunctionImport 元素 (CSDL)

概念性架构定义语言(CSDL)中的FunctionImport元素表示在数据源中定义但可通过概念模型用于对象的函数。例如，存储模型中的 Function 元素可用于表示数据库中的存储过程。概念模型中的FunctionImport元素表示实体框架应用程序中的相应函数并通过使用 FunctionImportMapping 元素映射到存储模型函数。在应用程序中调用函数时，会在数据库中执行相应的存储过程。

FunctionImport元素可以具有以下子元素(按所列顺序)：

- 文档(允许零个或一个元素)
- Parameter(允许零个或多个元素)
- Annotation 元素(允许零个或多个元素)
- ReturnType (FunctionImport)(允许零个或多个元素)

应为函数接受的每个参数定义一个参数元素。

函数的返回类型必须与returntype (FunctionImport)元素或returntype属性(见下文)一起指定，但不能同时指定两者。返回类型值必须是 EdmSimpleType、EntityType 或 ComplexType 的集合。

适用的属性

下表介绍可应用于FunctionImport元素的特性。

属性	必需	描述
■	是	导入的函数的名称。
ReturnType	是	函数返回的类型。如果函数未返回值，则不要使用此属性。否则，该值必须是 ComplexType、EntityType 或 EDMSimpleType 的集合。
EntitySet	是	如果函数返回实体类型的集合，则EntitySet的值必须是该集合所属的实体集。否则，不得使用 EntitySet 属性。
IsComposable	是	如果将该值设置为 true，则该函数是可组合的(表值函数)，可以在 LINQ 查询中使用。默认值为false。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于FunctionImport元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个FunctionImport元素，该元素接受一个参数并返回实体类型的集合：

```
<FunctionImport Name="GetStudentGrades"
    EntitySet="StudentGrade"
    ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Key 元素 (CSDL)

Key元素是EntityType元素的子元素，用于定义实体键(用于确定标识的实体类型的一个或一组属性)。构成实体键的属性是在设计时选择的。实体键属性的值必须在运行时唯一标识实体集中的实体类型实例。在选择构成实体键的属性时应确保实例在实体集中的唯一性。Key元素通过引用实体类型的一个或多个属性来定义实体键。

Key元素可以具有以下子元素：

- PropertyRef (一个或多个元素)
- 批注元素(零个或多个元素)

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于该键元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例定义了一个名为Book的实体类型。实体键通过引用实体类型的ISBN属性来定义。

```
<EntityType Name="Book">
    <Key>
        <PropertyRef Name="ISBN" />
    </Key>
    <Property Type="String" Name="ISBN" Nullable="false" />
    <Property Type="String" Name="Title" Nullable="false" />
    <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
    <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
        FromRole="Book" ToRole="Publisher" />
    <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
        FromRole="Book" ToRole="Author" />
</EntityType>
```

由于国际标准图书编号(ISBN)唯一标识书籍，因此ISBN属性是实体键的一个不错选择。

下面的示例演示一个实体类型(作者)，其中包含一个由两个属性、名称和地址组成的实体键。

```
<EntityType Name="Author">
    <Key>
        <PropertyRef Name="Name" />
        <PropertyRef Name="Address" />
    </Key>
    <Property Type="String" Name="Name" Nullable="false" />
    <Property Type="String" Name="Address" Nullable="false" />
    <NavigationProperty Name="Books" Relationship="BooksModel.WrittenBy"
        FromRole="Author" ToRole="Book" />
</EntityType>
```

使用实体键的名称和地址是合理的选择，因为同一名称的两个作者不会居住在同一地址。但是，针对实体键的这种选择并不能绝对确保实体键在实体集中的唯一性。在这种情况下，建议添加可用于唯一标识作者的属性，例如AuthorId：。

Member 元素 (CSDL)

Member元素是EnumType元素的子元素，用于定义枚举类型的成员。

适用的属性

下表介绍可应用于FunctionImport元素的特性。

属性	说明	示例
name	是	成员的名称。
value	是	成员的值。默认情况下，第一个成员的值为0，并且每个连续枚举数的值将增加1。可能存在多个具有相同值的成员。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于FunctionImport元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有三个成员元素的EnumType元素：

```
<EnumType Name="Color">
  <Member Name="Red" Value="1"/>
  <Member Name="Green" Value="3" />
  <Member Name="Blue" Value="5"/>
</EntityType>
```

NavigationProperty 元素 (CSDL)

NavigationProperty元素定义导航属性，该属性提供对关联的另一端的引用。与使用Property元素定义的属性不同，导航属性不定义数据的形状和特征。它们提供了一种在两个实体类型之间导航关联的方法。

注意，对于关联两端的两种实体类型，导航属性都是可选的。如果您对于位于关联一端的实体类型定义一个导航属性，则不需要对于位于关联另一端的实体类型定义导航属性。

导航属性返回的数据类型是由其远程关联端的重数确定的。例如，假设有一个导航属性OrdersNavProp，该属性存在于客户实体类型上，并在customer和Order之间导航一对多关联。因为导航属性的远程关联端的重数为many (*)，所以其数据类型为集合(顺序)。同样，如果“订单”实体类型上存在导航属性CustomerNavProp，则其数据类型为Customer，因为远程端的重数为一(1)。

NavigationProperty元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于**NavigationProperty**元素的特性。

属性	说明	示例
■	是	导航属性的名称。
■	是	处于模型的作用域中的关联的名称。
ToRole	是	导航在此结束的关联端。 ToRole 属性的值必须与某个关联端上定义的某个■属性(在 AssociationEnd 元素中定义)的值相同。
FromRole	是	导航从其开始的关联端。 FromRole 属性的值必须与某个关联端上定义的某个■属性(在 AssociationEnd 元素中定义)的值相同。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**NavigationProperty**元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例定义了一个具有两个导航属性(**PublishedBy**和**WrittenBy**)的实体类型(**Book**)：

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

OnDelete 元素 (CSDL)

概念架构定义语言(CSDL)中的**OnDelete**元素定义与关联关联的行为。如果在关联的一端将**Action**特性设置为**Cascade**, 则在删除第一条上的实体类型时, 将删除关联的另一端上的相关实体类型。如果两个实体类型之间的

关联是主键到主键关系，则当删除关联的另一端上的主体对象时，将删除加载的依赖对象，而不考虑OnDelete规范。

NOTE

OnDelete元素只影响应用程序的运行时行为；它不会影响数据源中的行为。在数据源中定义的行为应与在应用程序中定义的行为相同。

OnDelete元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于OnDelete元素的特性。

属性	说明	示例
Action	是	Cascade或None。如果在删除主体实体类型时将删除其的依赖实体类型，则为。如果为 "None"，则在删除主体实体类型时不会删除从属实体类型。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Association元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个定义CustomerOrders关联的association元素。OnDelete元素指示在删除客户时，将删除与特定客户相关且已加载到 ObjectContext 的所有订单。

```
<Association Name="CustomerOrders">
    <End Type="ExampleModel.Customer" Role="Customer" Multiplicity="1">
        <OnDelete Action="Cascade" />
    </End>
    <End Type="ExampleModel.Order" Role="Order" Multiplicity="*" />
</Association>
```

Parameter 元素 (CSDL)

概念架构定义语言(CSDL)中的Parameter元素可以是 FunctionImport 元素或 Function 元素的子元素。

FunctionImport 元素应用程序

Parameter元素(作为FunctionImport元素的子元素)用于定义在 CSDL 中声明的函数导入的输入和输出参数。

Parameter元素可以具有以下子元素(按所列顺序)：

- 文档(允许零个或一个元素)
- Annotation 元素(允许零个或多个元素)

适用的属性

下表介绍可应用于参数元素的特性。

■■■	■■■	■
■	是	参数的名称。
类型	是	参数类型。值必须为 EDMSimpleType 或模型范围内的复杂类型。
■	是	In 、 Out 或 InOut ，具体取决于参数是输入、输出还是输入/输出参数。
MaxLength	是	允许的参数最长长度。
■	是	参数的精度。
■	是	参数的确定位数。
SRID	是	空间系统引用标识符。仅对空间类型的参数有效。有关详细信息，请参阅 SRID and SRID (SQL Server) 。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有一个参数子元素的**FunctionImport**元素。函数接受一个输入参数并返回实体类型的集合。

```
<FunctionImport Name="GetStudentGrades"
    EntitySet="StudentGrade"
    ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

Function 元素应用程序

Parameter元素(作为**Function**元素的子元素)为在概念模型中定义或声明的函数定义参数。

Parameter元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- CollectionType (零个或一个元素)

- **ReferenceType** (零个或一个元素)
- **RowType** (零个或一个元素)

NOTE

只有一个**CollectionType**、**ReferenceType**或**RowType**元素可以是■元素的子元素。

- **Annotation** 元素(允许零个或多个元素)

NOTE

Annotation 元素必须出现在所有其他子元素之后。仅在 CSDL v2 和更高版本中允许使用批注元素。

适用的属性

下表介绍可应用于参数元素的特性。

■	■	■
■	是	参数的名称。
类型	是	参数类型。参数可以是以下任意类型(或这些类型的集合)： EdmSimpleType Entity Type — 实体类型 Complex Type — 复杂类型 行类型 Reference Type — 引用类型
■ Null	是	True (默认值)或False，具体取决于属性是否可以具有null值。
■	是	属性的默认值。
MaxLength	是	属性值的最大长度。
FixedLength	是	True或False，具体取决于属性值是否将作为固定长度字符串存储。
■	是	属性值的精度。
■	是	属性值的刻度。
SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息，请参阅 SRID and SRID (SQL Server) 。
Unicode	是	True或False，具体取决于属性值是否将存储为 Unicode 字符串。
■	是	指定要在数据源中使用的排序的字符串。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个使用一个参数子元素定义函数参数的函数元素。

```
<Function Name="GetYearsEmployed" ReturnType="Edm.Int32">
<Parameter Name="Instructor" Type="SchoolModel.Person" />
<DefiningExpression>
    Year(CurrentDateTime()) - Year(cast(Instructor.HireDate as DateTime))
</DefiningExpression>
</Function>
```

Principal 元素 (CSDL)

概念架构定义语言(CSDL)中的**主体元素**是 ReferentialConstraint 元素的子元素, 用于定义引用约束的主体端。ReferentialConstraint元素定义的功能与关系数据库中的引用完整性约束类似。与数据库表中的一个(或多个)列可以引用另一个表的主键相同, 实体类型的一个(或多个)属性可以引用另一个实体类型的实体键。引用的实体类型称为**约束的主体端**。引用主体端的实体类型称为**约束的依赖端**。PropertyRef元素用于指定依赖端引用的键。

主体元素可以具有以下子元素(按所列顺序):

- PropertyRef (一个或多个元素)
- 批注元素(零个或多个元素)

适用的属性

下表描述了可应用于主体元素的特性。

■	■	■
■	是	关联的主体端的实体类型的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例显示了ReferentialConstraint元素, 它是PublishedBy关联定义的一部分。发行者实体类型的Id属性构成了引用约束的主体端。

```

<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

Property 元素 (CSDL)

概念架构定义语言(CSDL)中的属性元素可以是 EntityType 元素、ComplexType 元素或 RowType 元素的子元素。

EntityType 和 ComplexType 元素应用程序

属性元素(作为EntityType或ComplexType元素的子元素)定义实体类型实例或复杂类型实例将包含的数据的形状和特征。概念模型中的属性类似于为类定义的属性。正如类的属性定义类的形状和携带有关对象的信息一样，概念模型中的属性也定义实体类型的形状和携带有关实体类型实例的信息。

Property元素可以具有以下子元素(按所列顺序)：

- 文档元素(允许零个或一个元素)
- Annotation 元素(允许零个或多个元素)

以下方面可应用于属性元素：可以为 null、默认值、MaxLength、FixedLength、精度、小数位数、Unicode、排序规则、ConcurrencyMode。方面是一些 XML 特性，它们提供有关如何在数据存储区中存储属性值的信息。

NOTE

Facet 只能应用于EDMSimpleType类型的属性。

适用的属性

下表介绍可应用于属性元素的特性。

属性	说明	示例
■ 名称	是	属性的名称。
类型	是	属性值的类型。属性值类型必须为 EDMSimpleType 或模型范围内的复杂类型(由完全限定名称指示)。
■ Null	是	True(默认值)或 False，取决于属性是否可以具有 null 值。 [!NOTE]

属性	说明	示例
CSDL v1 中的 <code>></code> ，复杂类型属性必须具有 <code>Nullable="False"</code> 。		
Default	是	属性的默认值。
MaxLength	是	属性值的最大长度。
FixedLength	是	True或False，具体取决于属性值是否将作为固定长度字符串存储。
Precision	是	属性值的精度。
Scale	是	属性值的刻度。
SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息，请参阅 SRID and SRID (SQL Server) 。
Unicode	是	True或False，具体取决于属性值是否将存储为 Unicode 字符串。
Collation	是	指定要在数据源中使用的排序的字符串。
ConcurrencyMode	是	None(默认值)或Optimistic。如果此值设置为Optimistic，会在乐观并发检查中使用属性值。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Property元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有三个属性元素的EntityType元素：

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

下面的示例演示一个包含五个属性元素的ComplexType元素：

```

<ComplexType Name="Address" >
  <Property Type="String" Name="StreetAddress" Nullable="false" />
  <Property Type="String" Name="City" Nullable="false" />
  <Property Type="String" Name="StateOrProvince" Nullable="false" />
  <Property Type="String" Name="Country" Nullable="false" />
  <Property Type="String" Name="PostalCode" Nullable="false" />
</ComplexType>

```

RowType 元素应用

Property元素(作为**RowType**元素的子元素)定义可传递到模型定义函数或从模型定义函数返回的数据的形状和特征。

Property元素可以具有以下子元素之一：

- **CollectionType**
- **ReferenceType**
- **RowType**

Property元素可以有任意数量的子批注元素。

NOTE

仅在 CSDL v2 和更高版本中允许使用批注元素。

适用的属性

下表介绍可应用于属性元素的特性。

属性	说明	示例
名称	是	属性的名称。
类型	是	属性值的类型。
Null	是	True(默认值)或 False, 取决于属性是否可以具有 null 值。 [!NOTE]
CSDL v1 中的 > 复杂类型属性必须具有 <code>Nullable="False"</code> 。		
默认值	是	属性的默认值。
MaxLength	是	属性值的最大长度。
FixedLength	是	True或False, 具体取决于属性值是否将作为固定长度字符串存储。
精度	是	属性值的精度。
刻度	是	属性值的刻度。

SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息, 请参阅 SRID and SRID (SQL Server) 。
Unicode	是	True或False, 具体取决于属性值是否将存储为 Unicode 字符串。
■	是	指定要在数据源中使用的排序的字符串。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Property元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了用于定义模型定义函数的返回类型的形状的属性元素。

```
<Function Name="LastNamesAfter">
  <Parameter Name="someString" Type="Edm.String" />
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="FirstName" Type="Edm.String" Nullable="false" />
        <Property Name="LastName" Type="Edm.String" Nullable="false" />
      </RowType>
    </CollectionType>
  </ReturnType>
  <DefiningExpression>
    SELECT VALUE ROW(p.FirstName, p.LastName)
    FROM SchoolEntities.People AS p
    WHERE p.LastName &gt;= somestring
  </DefiningExpression>
</Function>
```

PropertyRef 元素 (CSDL)

概念架构定义语言(CSDL)中的PropertyRef元素引用实体类型的属性, 以指示该属性将执行以下角色之一:

- 实体键的一部分(实体类型的用于确定标识的一个或一组属性)。一个或多个PropertyRef元素可用于定义实体键。
- 引用约束的依赖端或主体端。

PropertyRef元素只能将批注元素(零个或多个)作为子元素。

NOTE

仅在 CSDL v2 和更高版本中允许使用批注元素。

适用的属性

下表介绍可应用于PropertyRef元素的特性。

特性	说明	示例
■	是	所引用属性的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于PropertyRef元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例定义了一个实体类型(**Book**)。实体键通过引用实体类型的ISBN属性来定义。

```
<EntityType Name="Book">
  <Key>
    <PropertyRef Name="ISBN" />
  </Key>
  <Property Type="String" Name="ISBN" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" />
  <Property Type="Decimal" Name="Revision" Nullable="false" Precision="29" Scale="29" />
  <NavigationProperty Name="Publisher" Relationship="BooksModel.PublishedBy"
    FromRole="Book" ToRole="Publisher" />
  <NavigationProperty Name="Authors" Relationship="BooksModel.WrittenBy"
    FromRole="Book" ToRole="Author" />
</EntityType>
```

在下一个示例中, 两个PropertyRef元素用于指示两个属性(**Id**和**PublisherId**)是引用约束的主体端和依赖端。

```
<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

ReferenceType 元素 (CSDL)

概念架构定义语言(CSDL)中的**ReferenceType**元素指定对实体类型的引用。**ReferenceType**元素可以是以下元素的子元素：

- **ReturnType** (函数)
- 参数
- **CollectionType**

在定义函数的参数或返回类型时，将使用**ReferenceType**元素。

ReferenceType元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于**ReferenceType**元素的特性。

特性	说明	示例
类型	是	所引用的实体类型的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**ReferenceType**元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例显示了**ReferenceType**元素，该元素在接受对**Person**实体类型的引用的模型定义函数中用作**Parameter**元素的子元素：

```
<Function Name="GetYearsEmployed" ReturnType="Edm.Int32">
  <Parameter Name="instructor">
    <ReferenceType Type="SchoolModel.Person" />
  </Parameter>
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(cast(instructor.HireDate as DateTime))
  </DefiningExpression>
</Function>
```

下面的示例显示了**ReferenceType**元素，该元素用作模型定义的函数中的**ReturnType** (函数)元素的子元素，该函数返回对**Person**实体类型的引用：

```

<Function Name="GetPersonReference">
  <Parameter Name="p" Type="SchoolModel.Person" />
  <ReturnType>
    <ReferenceType Type="SchoolModel.Person" />
  </ReturnType>
  <DefiningExpression>
    REF(p)
  </DefiningExpression>
</Function>

```

ReferentialConstraint 元素 (CSDL)

概念架构定义语言(CSDL)中的ReferentialConstraint元素定义的功能与关系数据库中的引用完整性约束类似。与数据库表中的一个(或多个)列可以引用另一个表的主键相同，实体类型的一个(或多个)属性可以引用另一个实体类型的实体键。引用的实体类型称为约束的主体端。引用主体端的实体类型称为约束的依赖端。

如果一个实体类型上公开的外键引用另一个实体类型上的属性，则ReferentialConstraint元素将定义这两个实体类型之间的关联。由于ReferentialConstraint元素提供有关两个实体类型之间的关系的信息，因此，在映射规范语言(MSL)中不需要对应的AssociationSetMapping元素。没有公开外键的两个实体类型之间的关联必须具有相应的AssociationSetMapping元素，才能将关联信息映射到数据源。

如果在实体类型上没有公开外键，则ReferentialConstraint元素只能定义实体类型和其他实体类型之间的主键到主键约束。

ReferentialConstraint元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- Principal (恰好一个元素)
- Dependent(恰好一个元素)
- 批注元素(零个或多个元素)

适用的属性

ReferentialConstraint元素可以具有任意数量的批注特性(自定义 XML 特性)。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了在PublishedBy关联的定义中使用的ReferentialConstraint元素。

```

<Association Name="PublishedBy">
  <End Type="BooksModel.Book" Role="Book" Multiplicity="*" />
  </End>
  <End Type="BooksModel.Publisher" Role="Publisher" Multiplicity="1" />
  <ReferentialConstraint>
    <Principal Role="Publisher">
      <PropertyRef Name="Id" />
    </Principal>
    <Dependent Role="Book">
      <PropertyRef Name="PublisherId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

ReturnType (Function) 元素 (CSDL)

概念架构定义语言(CSDL)中的**ReturnType** (函数)元素指定在 `function` 元素中定义的函数的返回类型。还可以使用**ReturnType**特性指定函数返回类型。

返回类型可以是任何**EdmSimpleType**、实体类型、复杂类型、行类型、引用类型或这些类型之一的集合。

函数的返回类型可以使用**ReturnType** (function)元素的**type**属性指定，也可以使用以下子元素之一指定：

- `CollectionType`
- `ReferenceType`
- `RowType`

NOTE

如果指定的函数返回类型同时具有**ReturnType** (函数)元素的**type**属性和一个子元素，模型将不会验证。

适用的属性

下表介绍可应用于**ReturnType** (函数)元素的特性。

特性	说明	示例
<code>ReturnType</code>	是	函数返回的类型。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**ReturnType** (Function)元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例使用`function`元素来定义一个函数，该函数返回书籍已打印的年数。请注意，返回类型由**ReturnType** (函数)元素的**type**属性指定。

```
<Function Name="GetYearsInPrint">
  <ReturnType Type=="Edm.Int32">
    <Parameter Name="book" Type="BooksModel.Book" />
    <DefiningExpression>
      Year(CurrentDateTime()) - Year(cast(book.PublishedDate as DateTime))
    </DefiningExpression>
  </ReturnType>
</Function>
```

ReturnType (FunctionImport) 元素 (CSDL)

概念架构定义语言(CSDL)中的**ReturnType** (**FunctionImport**)元素指定在 **FunctionImport** 元素中定义的函数的返回类型。还可以使用**ReturnType**特性指定函数返回类型。

返回类型可以是实体类型、复杂类型或**EdmSimpleType**的任何集合。

函数的返回类型是使用**ReturnType** (**FunctionImport**)元素的**type**属性指定的。

适用的属性

下表介绍可应用于**ReturnType** (**FunctionImport**)元素的特性。

特性	说明	示例
类型	是	函数返回的类型。该值必须是 ComplexType 、 EntityType 或 EDMSimpleType 的集合。
EntitySet	是	如果函数返回实体类型的集合，则 EntitySet 的值必须是该集合所属的实体集。否则，不得使用 EntitySet 属性。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**ReturnType** (**FunctionImport**)元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例使用返回书籍和出版商的**FunctionImport**。请注意，函数将返回两个结果集，因而指定了两个**ReturnType** (**FunctionImport**)元素。

```
<FunctionImport Name="GetBooksAndPublishers">
  <ReturnType Type=="Collection(BooksModel.Book )" EntitySet="Books">
  <ReturnType Type=="Collection(BooksModel.Publisher)" EntitySet="Publishers">
</FunctionImport>
```

RowType 元素 (CSDL)

概念架构定义语言(CSDL)中的**RowType**元素将未命名的结构定义为概念模型中定义的函数的参数或返回类型。

RowType元素可以是以下元素的子元素：

- **CollectionType**
- **参数**
- **ReturnType** (函数)

RowType元素可以具有以下子元素(按所列顺序)：

- **Property**(一个或多个)
- 批注元素(零个或多个)

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于RowType元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个模型定义函数, 该函数使用CollectionType元素来指定该函数返回行的集合(在RowType元素中指定)。

```
<Function Name="LastNamesAfter">
  <Parameter Name="someString" Type="Edm.String" />
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="FirstName" Type="Edm.String" Nullable="false" />
        <Property Name="LastName" Type="Edm.String" Nullable="false" />
      </RowType>
    </CollectionType>
  </ReturnType>
  <DefiningExpression>
    SELECT VALUE ROW(p.FirstName, p.LastName)
    FROM SchoolEntities.People AS p
    WHERE p.LastName &gt;= somestring
  </DefiningExpression>
</Function>
```

Schema 元素 (CSDL)

Schema元素是概念模型定义的根元素。它包括构成概念模型的对象、函数和容器的定义。

Schema元素可包含零个或多个下列子元素:

- 使用
- EntityContainer
- EntityType
- EnumType
- 关联
- ComplexType
- 函数

一个Schema元素可包含零个或一个批注元素。

NOTE

仅在 CSDL v2 和更高版本中允许Function元素和 annotation 元素。

Schema元素使用namespace特性为概念模型中的实体类型、复杂类型和关联对象定义命名空间。在命名空间内, 任何两个对象都不能同名。命名空间可以跨多个架构元素和多个 csdl 文件。

概念模型命名空间与Schema元素的 XML 命名空间不同。概念模型命名空间(由命名空间特性定义)是实体类型、复杂类型和关联类型的逻辑容器。Schema元素的 XML 命名空间(由xmlns特性指示)是schema元素的子元素和属性的默认命名空间。格式为 <https://schemas.microsoft.com/ado/YYYY/MM/edm> 的 XML 命名空间(其中, YYYY 和 MM 分别表示年份和月份)是为 CSDL 保留的。自定义元素和特性不能位于具有此格式的命名空间中。

适用的属性

下表描述了可对Schema元素应用的属性。

属性	说明	示例
Namespace	是	概念模型的命名空间。■属性的值用于构成类型的完全限定名称。例如，如果名为Customer的entitytype位于Simple 命名空间中，则entitytype的完全限定名称为simpleexamplemodel.customer。以下字符串不能用作Namespace特性的值：System、■或Edm。■属性的值不能与 SSDL 架构元素中namespace属性的值相同。
Alias	是	用于取代命名空间名称的标识符。例如，如果名为Customer的EntityType位于 Simple 命名空间，并且Alias属性的值为Model，则可以使用 model 作为 EntityType 的完全限定名称 ■

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Schema元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个架构元素，该元素包含一个EntityContainer元素、两个EntityType元素和一个Association元素。

```

<Schema xmlns="https://schemas.microsoft.com/ado/2009/11/edm"
    xmlns:cg="https://schemas.microsoft.com/ado/2009/11/codegeneration"
    xmlns:store="https://schemas.microsoft.com/ado/2009/11/edm/EntityStoreSchemaGenerator"
    Namespace="ExampleModel" Alias="Self">
    <EntityTypeContainer Name="ExampleModelContainer">
        <EntityTypeSet Name="Customers"
            EntityType="ExampleModel.Customer" />
        <EntityTypeSet Name="Orders" EntityType="ExampleModel.Order" />
        <AssociationSet
            Name="CustomerOrder"
            Association="ExampleModel.CustomerOrders">
            <End Role="Customer" EntitySet="Customers" />
            <End Role="Order" EntitySet="Orders" />
        </AssociationSet>
    </EntityTypeContainer>
    <EntityType Name="Customer">
        <Key>
            <PropertyRef Name="CustomerId" />
        </Key>
        <Property Type="Int32" Name="CustomerId" Nullable="false" />
        <Property Type="String" Name="Name" Nullable="false" />
        <NavigationProperty
            Name="Orders"
            Relationship="ExampleModel.CustomerOrders"
            FromRole="Customer" ToRole="Order" />
    </EntityType>
    <EntityType Name="Order">
        <Key>
            <PropertyRef Name="OrderId" />
        </Key>
        <Property Type="Int32" Name="OrderId" Nullable="false" />
        <Property Type="Int32" Name="ProductId" Nullable="false" />
        <Property Type="Int32" Name="Quantity" Nullable="false" />
        <NavigationProperty
            Name="Customer"
            Relationship="ExampleModel.CustomerOrders"
            FromRole="Order" ToRole="Customer" />
        <Property Type="Int32" Name="CustomerId" Nullable="false" />
    </EntityType>
    <Association Name="CustomerOrders">
        <End Type="ExampleModel.Customer"
            Role="Customer" Multiplicity="1" />
        <End Type="ExampleModel.Order"
            Role="Order" Multiplicity="*" />
        <ReferentialConstraint>
            <Principal Role="Customer">
                <PropertyRef Name="CustomerId" />
            </Principal>
            <Dependent Role="Order">
                <PropertyRef Name="CustomerId" />
            </Dependent>
        </ReferentialConstraint>
    </Association>
</Schema>

```

TypeRef 元素 (CSDL)

概念架构定义语言(CSDL)中的TypeRef元素提供对现有命名类型的引用。TypeRef元素可以是 CollectionType 元素的子元素，用于指定函数具有集合作为参数或返回类型。

TypeRef元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- 批注元素(零个或多个元素)

适用的属性

下表介绍可应用于TypeRef元素的特性。请注意，**DefaultValue**、**MaxLength**、**FixedLength**、**精度**、**小数位数**、**Unicode**和**排序规则**特性仅适用于edmsimpletype。

属性	说明	值
类型	是	所引用的类型的名称。
■ Null	是	True (默认值)或 False , 取决于属性是否可以具有 null 值。 [!NOTE]
CSDL v1 中的 > 复杂类型属性必须具有 <code>Nullable="False"</code> 。		
■	是	属性的默认值。
MaxLength	是	属性值的最大长度。
FixedLength	是	True 或 False , 具体取决于属性值是否将作为固定长度字符串存储。
■	是	属性值的精度。
■	是	属性值的刻度。
SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息, 请参阅 SRID (SQL Server) 。
Unicode	是	True 或 False , 具体取决于属性值是否将存储为 Unicode 字符串。
■	是	指定要在数据源中使用的排序的字符串。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于CollectionType元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个模型定义函数, 该函数使用TypeRef元素(作为CollectionType元素的子元素)来指定该函数接受部门实体类型的集合。

```

<Function Name="GetAvgBudget">
  <Parameter Name="Departments">
    <CollectionType>
      <TypeRef Type="SchoolModel.Department"/>
    </CollectionType>
  </Parameter>
  <ReturnType Type="Collection(Edm.Decimal)"/>
  <DefiningExpression>
    SELECT VALUE AVG(d.Budget) FROM Departments AS d
  </DefiningExpression>
</Function>

```

Using 元素 (CSDL)

概念性架构定义语言(CSDL)中的Using元素导入存在于其他命名空间中的概念模型的内容。通过设置Namespace属性的值，您可以引用在其他概念模型中定义的实体类型、复杂类型和关联类型。多个Using元素可以是Schema元素的子元素。

NOTE

CSDL 中的using元素的作用与编程语言中的using语句完全相同。通过使用编程语言中的using语句导入命名空间，不会影响原始命名空间中的对象。在 CSDL 中，导入的命名空间可以包含从原始命名空间中的实体类型导出的实体类型。这可能影响在原始命名空间中声明的实体集。

Using元素可以具有以下子元素：

- 文档(允许零个或一个元素)
- Annotation 元素(允许零个或多个元素)

适用的属性

下表描述了可应用于Using元素的特性。

属性	说明	示例
Namespace	是	导入的命名空间的名称。
Alias	是	用于取代命名空间名称的标识符。尽管此特性是必需的，但并不要求使用它来取代命名空间以限定对象名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Using元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示用于导入在其他位置定义的命名空间的Using元素。请注意，显示的架构元素的命名空间是BooksModel的。Publisher EntityType上的Address属性是在ExtendedBooksModel命名空间中定义的复杂类型（使用Using元素导入）。

```
<Schema xmlns="https://schemas.microsoft.com/ado/2009/11/edm"
    xmlns:cg="https://schemas.microsoft.com/ado/2009/11/codegeneration"
    xmlns:store="https://schemas.microsoft.com/ado/2009/11/edm/EntityStoreSchemaGenerator"
    Namespace="BooksModel" Alias="Self">

    <Using Namespace="BooksModel.Extended" Alias="BMEExt" />

    <EntityContainer Name="BooksContainer" >
        <EntitySet Name="Publishers" EntityType="BooksModel.Publisher" />
    </EntityContainer>

    <EntityType Name="Publisher">
        <Key>
            <PropertyRef Name="Id" />
        </Key>
        <Property Type="Int32" Name="Id" Nullable="false" />
        <Property Type="String" Name="Name" Nullable="false" />
        <Property Type="BMEExt.Address" Name="Address" Nullable="false" />
    </EntityType>

</Schema>
```

Annotation 特性 (CSDL)

以概念架构定义语言 (CSDL) 表示的批注特性是概念模型中的自定义 XML 特性。除了具有有效的 XML 结构之外，还必须满足批注特性的以下各项条件：

- 批注特性不能位于为 CSDL 保留的任何 XML 命名空间中。
- 可以将多个批注特性应用于给定的 CSDL 元素。
- 任何两个批注特性的完全限定名称都不能相同。

可以使用批注特性提供有关概念模型中元素的额外元数据。可以在运行时通过使用 System.web 命名空间中的类访问批注元素中包含的元数据。

示例

下面的示例演示具有批注特性(CustomAttribute)的EntityType元素。本示例还演示了一个应用于实体类型元素的批注元素。

```

<Schema Namespace="SchoolModel" Alias="Self"
    xmlns:annotation="https://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="https://schemas.microsoft.com/ado/2009/11/edm">
    <EntityContainer Name="SchoolEntities" annotation:LazyLoadingEnabled="true">
        <EntityType Name="Person" EntityType="SchoolModel.Person" />
    </EntityContainer>
    <EntityType Name="Person" xmlns:p="http://CustomNamespace.com"
        p:CustomAttribute="Data here.">
        <Key>
            <PropertyRef Name="PersonID" />
        </Key>
        <Property Name="PersonID" Type="Int32" Nullable="false"
            annotation:StoreGeneratedPattern="Identity" />
        <Property Name="LastName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="FirstName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="HireDate" Type="DateTime" />
        <Property Name="EnrollmentDate" Type="DateTime" />
        <p:CustomElement>
            Custom metadata.
        </p:CustomElement>
    </EntityType>
</Schema>

```

下面的代码检索批注特性中的元数据并将其写入控制台：

```

EdmItemCollection collection = new EdmItemCollection("School.csdl");
MetadataWorkspace workspace = new MetadataWorkspace();
workspace.RegisterItemCollection(collection);
EdmType contentType;
workspace.TryGetType("Person", "SchoolModel", DataSpace.CSpace, out contentType);
if (contentType.MetadataProperties.Contains("http://CustomNamespace.com:CustomAttribute"))
{
    MetadataProperty annotationProperty =
        contentType.MetadataProperties["http://CustomNamespace.com:CustomAttribute"];
    object annotationValue = annotationProperty.Value;
    Console.WriteLine(annotationValue.ToString());
}

```

上面的代码假定 `School.csdl` 文件位于项目的输出目录中，且您已将下面的 `Imports` 和 `Using` 语句添加到项目中：

```
using System.Data.Metadata.Edm;
```

Annotation 元素 (CSDL)

以概念架构定义语言 (CSDL) 表示的批注元素是概念模型中的自定义 XML 元素。除了具有有效的 XML 结构之外，还必须满足批注元素的以下各项条件：

- 批注元素不能位于为 CSDL 保留的任何 XML 命名空间中。
- 多个批注元素可能是某个给定 CSDL 元素的子元素。

- 任何两个批注元素的完全限定名称都不能相同。
- 批注元素必须出现在给定 CSDL 元素的所有其他子元素之后。

可以使用批注元素提供有关概念模型中元素的额外元数据。从 .NET Framework 版本 4 开始，可以在运行时通过使用 System.web 命名空间中的类访问批注元素中包含的元数据。

示例

下面的示例显示具有 annotation 元素(CustomElement)的EntityType元素。本示例还演示了一个应用于实体类型元素的批注特性。

```
<Schema Namespace="SchoolModel" Alias="Self"
    xmlns:annotation="https://schemas.microsoft.com/ado/2009/02/edm/annotation"
    xmlns="https://schemas.microsoft.com/ado/2009/11/edm">
    <EntityContainer Name="SchoolEntities" annotation:LazyLoadingEnabled="true">
        <EntitySet Name="People" EntityType="SchoolModel.Person" />
    </EntityContainer>
    <EntityType Name="Person" xmlns:p="http://CustomNamespace.com"
        p:CustomAttribute="Data here.">
        <Key>
            <PropertyRef Name="PersonID" />
        </Key>
        <Property Name="PersonID" Type="Int32" Nullable="false"
            annotation:StoreGeneratedPattern="Identity" />
        <Property Name="LastName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="FirstName" Type="String" Nullable="false"
            MaxLength="50" Unicode="true" FixedLength="false" />
        <Property Name="HireDate" Type="DateTime" />
        <Property Name="EnrollmentDate" Type="DateTime" />
        <p:CustomElement>
            Custom metadata.
        </p:CustomElement>
    </EntityType>
</Schema>
```

下面的代码检索批注元素中的元数据并将其写入控制台：

```
EdmItemCollection collection = new EdmItemCollection("School.csdl");
MetadataWorkspace workspace = new MetadataWorkspace();
workspace.RegisterItemCollection(collection);
EdmType contentType;
workspace.TryGetType("Person", "SchoolModel", DataSpace.CSpace, out contentType);
if (contentType.MetadataProperties.Contains("http://CustomNamespace.com:CustomElement"))
{
    MetadataProperty annotationProperty =
        contentType.MetadataProperties["http://CustomNamespace.com:CustomElement"];
    object annotationValue = annotationProperty.Value;
    Console.WriteLine(annotationValue.ToString());
}
```

上面的代码假定 School.csdl 文件位于项目的输出目录中并且您已将下面的 Imports 和 Using 语句添加到项目中：

```
using System.Data.Metadata.Edm;
```

概念模型类型 (CSDL)

概念性架构定义语言(CSDL)支持一组抽象基元数据类型(称为`edmsimpletype`), 用于定义概念模型中的属性。`Edmsimpletype`是存储或承载环境中支持的基元数据类型的代理。

下表列出了 CSDL 支持的基元数据类型。该表还列出了可应用于每个EDMSimpleType的分面。

EDMSIMPLETYPE	分面	分面
<code>Edm</code>	包含二进制数据。	<code>MaxLength</code> 、 <code>FixedLength</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.Boolean</code>	包含值 <code>true</code> 或 <code>false</code> 。	<code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含一个无符号的 8 位整数值。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.DateTime</code>	表示日期和时间。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.DateTimeOffset</code>	包含日期和时间, 采用相对于 GMT 的偏移量(以分钟为单位)的形式。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含一个具有固定精度和小数位数的数据。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.Double</code>	包含具有15位精度的浮点数	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含一个具有 7 位精度的浮点数。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.Guid</code>	包含一个 16 字节的唯一标识符。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含一个带符号的 16 位整数值。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.Int32</code>	包含一个带符号的 32 位整数值。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.Int64</code>	包含一个带符号的 64 位整数值。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含一个带符号的 8 位整数值。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm.String</code>	包含字符数据。	<code>Unicode</code> 、 <code>FixedLength</code> 、 <code>MaxLength</code> 、 <code>Collation</code> 、 <code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>	包含当天的时间。	<code>Precision</code> 、 <code>Nullable</code> 、 <code>Default</code>
<code>Edm</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>
<code>Edm.GeographyPoint</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>
<code>Edm.GeographyLineString</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>
<code>Edm.GeographyPolygon</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>
<code>Edm.GeographyMultiPoint</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>
<code>Edm.GeographyMultiLineString</code>		<code>Nullable</code> 、 <code>Default</code> 、 <code>SRID</code>

EDMSIMPLETYPE	¶	
Edm. GeographyMultiPolygon		Nullable、Default、SRID
Edm. GeographyCollection		Nullable、Default、SRID
Edm		Nullable、Default、SRID
Edm. GeometryPoint		Nullable、Default、SRID
Edm. GeometryLineString		Nullable、Default、SRID
Edm. GeometryPolygon		Nullable、Default、SRID
Edm. GeometryMultiPoint		Nullable、Default、SRID
Edm. GeometryMultiLineString		Nullable、Default、SRID
Edm. GeometryMultiPolygon		Nullable、Default、SRID
Edm. GeometryCollection		Nullable、Default、SRID

方面 (CSDL)

以概念架构定义语言 (CSDL) 表示的方面表示对于实体类型和复杂类型的属性的约束。方面作为 XML 特性出现在以下 CSDL 元素上：

- 属性
- TypeRef
- 参数

下表描述了 CSDL 中支持的方面。所有方面都是可选的。从概念模型生成数据库时，实体框架将使用下面列出的一些方面。

NOTE

有关概念模型中的数据类型的信息，请参阅概念模型类型 (CSDL)。

¶	¶			
■	指定在对属性值执行比较和排序操作时要使用的排序序列。	Edm.String	是	是
ConcurrencyMode	表示应使用属性的值来进行开放式并发检查。	所有 EDMSimpleType 属性	是	是
Default	如果在安装时未提供值，则指定属性的默认值。	所有 EDMSimpleType 属性	是	是

FixedLength	指定属性值的长度是否可变。	Edm、edm	是	是
MaxLength	指定属性值的最大长度。	Edm、edm	是	是
■ Null	指定属性是否可以具有null值。	所有EDMSimpleType属性	是	是
■	对于Decimal类型的属性, 指定属性值可以具有的位数。对于类型为Time、DateTime和DateTimeOffset的属性, 指定属性值的秒的小数部分的位数。	Edm、DateTime、edm、edm、edm、Time	是	是
■	指定属性值小数点右侧的位数。	Edm	是	是
SRID	指定空间系统引用系统 ID。有关详细信息, 请参阅 SRID and SRID (SQL Server) 。	Edm、GeographyPoint、GeographyLineString、GeographyPolygon、GeographyMultiPoint、GeographyMultiLineString、GeographyMultiPolygon、edm、GeographyCollection、GeometryPoint、GeometryLineString、GeometryPolygon、GeometryMultiPoint、GeometryMultiLineString、GeometryMultiPolygon、GeometryCollection	是	是
Unicode	指示是否将属性值存储为 Unicode。	Edm.String	是	是

NOTE

当从概念模型生成数据库时, "生成数据库" 向导将识别属性元素上的StoreGeneratedPattern属性的值(如果该■位于以下命名空间中): <https://schemas.microsoft.com/ado/2009/02/edm/annotation>。特性支持的值为Identity和■。■值将生成一个数据库列, 其中包含在数据库中生成的标识值。■的值将生成一个列, 该列具有在数据库中计算的值。

示例

下面的示例演示了应用于实体类型的属性的方面：

```
<EntityType Name="Product">
  <Key>
    <PropertyRef Name="ProductId" />
  </Key>
  <Property Type="Int32"
    Name="ProductId" Nullable="false"
    a:StoreGeneratedPattern="Identity"
    xmlns:a="https://schemas.microsoft.com/ado/2009/02/edm/annotation" />
  <Property Type="String"
    Name="ProductName"
    Nullable="false"
    MaxLength="50" />
  <Property Type="String"
    Name="Location"
    Nullable="true"
    MaxLength="25" />
</EntityType>
```

MSL 规范

2020/3/11 ·

映射规范语言 (MSL) 是一种基于 XML 的语言，用于描述实体框架应用程序的概念模型和存储模型之间的映射。

在实体框架应用程序中，将在生成时从 msl 文件（用 MSL 编写）加载映射元数据。实体框架在运行时使用映射元数据将针对概念模型的查询转换为特定于存储的命令。

Entity Framework Designer (EF 设计器) 在设计时将映射信息存储在 .edmx 文件中。在生成时，Entity Designer 使用 .edmx 文件中的信息创建实体框架在运行时所需的 msl 文件。

MSL 中引用的所有概念模型类型或存储模型类型的名称必须由其各自的命名空间名称限定。有关概念模型命名空间名称的信息，请参阅 [CSDL 规范](#)。有关存储模型命名空间名称的信息，请参阅 [SSDL 规范](#)。

MSL 版本由 XML 命名空间进行区分。

MSL 版本	XML 命名空间
MSL v1	urn:架构-microsoft-com: windows: storage: 映射: CS
MSL v2	https://schemas.microsoft.com/ado/2008/09/mapping/cs
MSL v3	https://schemas.microsoft.com/ado/2009/11/mapping/cs

Alias 元素 (MSL)

映射规范语言 (MSL) 中的 Alias 元素是映射元素的子元素，用于定义概念模型和存储模型命名空间的别名。MSL 中引用的所有概念模型类型或存储模型类型的名称必须由其各自的命名空间名称限定。有关概念模型命名空间名称的信息，请参阅 Schema 元素 (CSDL)。有关存储模型命名空间名称的信息，请参阅 Schema 元素 (SSDL)。

Alias 元素不能有子元素。

适用的属性

下表描述了可应用于 Alias 元素的特性。

特性	说明	示例
Key	是	该特性指定的命名空间的别名。
	是	该元素的值为其别名的命名空间。

示例

下面的示例演示一个 alias 元素，该元素为在概念模型中定义的类型定义别名 `c`。

```

<Mapping Space="C-S"
  xmlns="https://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
      <MappingFragment StoreEntitySet="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="Credits" ColumnName="Credits" />
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="c.Department">
    <MappingFragment StoreEntitySet="Department">
      <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      <ScalarProperty Name="Name" ColumnName="Name" />
      <ScalarProperty Name="Budget" ColumnName="Budget" />
      <ScalarProperty Name="StartDate" ColumnName="StartDate" />
      <ScalarProperty Name="Administrator" ColumnName="Administrator" />
    </MappingFragment>
  </EntityTypeMapping>
  </EntityTypeMapping>
</EntityContainerMapping>
</Mapping>

```

AssociationEnd 元素 (MSL)

当概念模型中的实体类型的修改函数映射到基础数据库中的存储过程时，将使用映射规范语言 (MSL) 中的**AssociationEnd**元素。如果修改存储过程所使用的参数的值保存在关联属性中，则**AssociationEnd**元素会将属性值映射到该参数。有关更多信息，请参见下面的示例。

有关将实体类型的修改函数映射到存储过程的详细信息，请参阅 [ModificationFunctionMapping 元素 \(MSL\)](#) 和演练：将实体映射到存储过程。

AssociationEnd元素可以具有以下子元素：

- [ScalarProperty](#)

适用的属性

下表描述了适用于**AssociationEnd**元素的特性。

属性	说明	示例
AssociationSet	是	要映射的关联的名称。
From	是	导航属性的 FromRole 属性的值，该属性对应于要映射的关联。有关详细信息，请参阅 NavigationProperty 元素 (CSDL) 。
To	是	导航属性的 ToRole 属性的值，该属性对应于要映射的关联。有关详细信息，请参阅 NavigationProperty 元素 (CSDL) 。

示例

请考虑使用以下概念模型实体类型：

```

<EntityType Name="Course">
  <Key>
    <PropertyRef Name="CourseID" />
  </Key>
  <Property Type="Int32" Name="CourseID" Nullable="false" />
  <Property Type="String" Name="Title" Nullable="false" MaxLength="100"
    FixedLength="false" Unicode="true" />
  <Property Type="Int32" Name="Credits" Nullable="false" />
  <NavigationProperty Name="Department"
    Relationship="SchoolModel.FK_Course_Department"
    FromRole="Course" ToRole="Department" />
</EntityType>

```

另请考虑使用以下存储过程：

```

CREATE PROCEDURE [dbo].[UpdateCourse]
  @CourseID int,
  @Title nvarchar(50),
  @Credits int,
  @DepartmentID int
AS
  UPDATE Course SET Title=@Title,
    Credits=@Credits,
    DepartmentID=@DepartmentID
  WHERE CourseID=@CourseID;

```

若要将 **Course** 实体的更新函数映射到此存储过程，必须为**DepartmentID**参数提供一个值。**DepartmentID** 的值与实体类型中的某个属性不对应；该值包含在独立的关联中，此关联的映射如下所示：

```

<AssociationSetMapping Name="FK_Course_Department"
  TypeName="SchoolModel.FK_Course_Department"
  StoreEntitySet="Course">
  <EndProperty Name="Course">
    <ScalarProperty Name="CourseID" ColumnName="CourseID" />
  </EndProperty>
  <EndProperty Name="Department">
    <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
  </EndProperty>
</AssociationSetMapping>

```

下面的代码演示了**AssociationEnd**元素，该元素用于将**FK_课程_部门**关联的**DepartmentID**属性映射到**UpdateCourse**存储过程（**课程**实体类型的更新函数映射到该存储过程）：

```

<EntitySetMapping Name="Courses">
  <EntityTypeMapping TypeName="SchoolModel.Course">
    <MappingFragment StoreEntitySet="Course">
      <ScalarProperty Name="Credits" ColumnName="Credits" />
      <ScalarProperty Name="Title" ColumnName="Title" />
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="SchoolModel.Course">
    <ModificationFunctionMapping>
      <UpdateFunction FunctionName="SchoolModel.Store.UpdateCourse">
        <AssociationEnd AssociationSet="FK_Course_Department"
          From="Course" To="Department">
          <ScalarProperty Name="DepartmentID"
            ParameterName="DepartmentID"
            Version="Current" />
        </AssociationEnd>
        <ScalarProperty Name="Credits" ParameterName="Credits"
          Version="Current" />
        <ScalarProperty Name="Title" ParameterName="Title"
          Version="Current" />
        <ScalarProperty Name="CourseID" ParameterName="CourseID"
          Version="Current" />
      </UpdateFunction>
    </ModificationFunctionMapping>
  </EntityTypeMapping>
</EntitySetMapping>

```

AssociationSetMapping 元素 (MSL)

映射规范语言 (MSL) 中的 **AssociationSetMapping** 元素定义概念模型中的关联与基础数据库中的表列之间的映射。

概念模型中的关联是一些属性表示基础数据库中的主键列和外键列的类型。**AssociationSetMapping** 元素使用两个 **EndProperty** 元素来定义关联类型属性与数据库中的列之间的映射。您可以使用 **Condition** 元素对这些映射施加条件。使用 **ModificationFunctionMapping** 元素将关联的插入、更新和删除函数映射到数据库中的存储过程。通过在 **QueryView** 元素中使用实体 SQL 字符串，定义 association 和表列之间的只读映射。

NOTE

如果为概念模型中的关联定义了引用约束，则不需要将关联映射到 **AssociationSetMapping** 元素。如果具有引用约束的关联存在 **AssociationSetMapping** 元素，则将忽略在 **AssociationSetMapping** 元素中定义的映射。有关详细信息，请参阅 **ReferentialConstraint** 元素 (CSDL)。

AssociationSetMapping 元素可以具有以下子元素

- **QueryView** (零个或一个)
- **EndProperty** (零个或两个)
- 条件 (零个或多个)
- **ModificationFunctionMapping** (零个或一个)

适用的属性

下表介绍可应用于 **AssociationSetMapping** 元素的特性。

属性	说明	示例
■ Name	是	要映射的概念模型关联集的名称。

属性	说明	示例
TypeName	是	要映射的概念模型关联类型的命名空间限定的名称。
StoreEntitySet	是	要映射的表的名称。

示例

下面的示例显示一个AssociationSetMapping元素，在该元素中，在概念模型中设置的FK_课程_部门关联将映射到数据库中的课程表。在子EndProperty元素中指定关联类型属性与表列之间的映射。

```
<AssociationSetMapping Name="FK_Course_Department"
    TypeName="SchoolModel.FK_Course_Department"
    StoreEntitySet="Course">
    <EndProperty Name="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
</AssociationSetMapping>
```

ComplexProperty 元素 (MSL)

映射规范语言 (MSL) 中的ComplexProperty元素定义概念模型实体类型上的复杂类型属性与基础数据库中的表列之间的映射。属性-列映射在子 ScalarProperty 元素中指定。

ComplexType属性元素可以具有以下子元素：

- ScalarProperty (零个或多个)
- ComplexProperty (零个或多个)
- ComplexTypeMapping (零个或多个)
- 条件 (零个或多个)

适用的属性

下表描述了适用于ComplexProperty元素的属性：

属性	说明	示例
■	是	概念模型中要映射的实体类型的复杂属性的名称。
TypeName	是	概念模型属性类型的命名空间限定名称。

示例

下面的示例基于 School 模型。下面的复杂类型已添加到概念模型中：

```

<ComplexType Name="FullName">
    <Property Type="String" Name="LastName"
        Nullable="false" MaxLength="50"
        FixedLength="false" Unicode="true" />
    <Property Type="String" Name="FirstName"
        Nullable="false" MaxLength="50"
        FixedLength="false" Unicode="true" />
</ComplexType>

```

Person 实体类型的 LastName 和 FirstName 属性已替换为一个复杂属性名称：

```

<EntityType Name="Person">
    <Key>
        <PropertyRef Name="PersonID" />
    </Key>
    <Property Name="PersonID" Type="Int32" Nullable="false"
        annotation:StoreGeneratedPattern="Identity" />
    <Property Name="HireDate" Type="DateTime" />
    <Property Name="EnrollmentDate" Type="DateTime" />
    <Property Name="Name" Type="SchoolModel.FullName" Nullable="false" />
</EntityType>

```

以下 MSL 显示了用于将 Name 属性映射到基础数据库中的列的 ComplexProperty 元素：

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate" ColumnName="EnrollmentDate" />
            <ComplexProperty Name="Name" TypeName="SchoolModel.FullName">
                <ScalarProperty Name="FirstName" ColumnName="FirstName" />
                <ScalarProperty Name="LastName" ColumnName="LastName" />
            </ComplexProperty>
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

ComplexTypeMapping 元素 (MSL)

映射规范语言 (MSL) 中的 ComplexTypeMapping 元素是 ResultMapping 元素的子元素，在以下条件成立时定义概念模型中的函数导入与基础数据库中的存储过程之间的映射：

- 函数导入返回一个概念复杂类型。
- 存储过程返回的列的名称与复杂类型的属性名称不完全匹配。

默认情况下，存储过程返回的列与复杂类型返回的列之间的映射基于列名称和属性名称。如果列名称与属性名称不完全匹配，则必须使用 ComplexTypeMapping 元素来定义映射。有关默认映射的示例，请参阅 FunctionImportMapping 元素 (MSL)。

ComplexTypeMapping 元素可以具有以下子元素：

- ScalarProperty (零个或多个)

适用的属性

下表描述了适用于 ComplexTypeMapping 元素的特性。

TypeName	是	要映射的复杂类型的命名空间限定的名称。

示例

请思考以下存储过程：

```
CREATE PROCEDURE [dbo].[GetGrades]
    @student_Id int
    AS
        SELECT EnrollmentID as enroll_id,
               Grade as grade,
               CourseID as course_id,
               StudentID as student_id
        FROM dbo.StudentGrade
        WHERE StudentID = @student_Id
```

另请考虑使用以下概念模型复杂类型：

```
<ComplexType Name="GradeInfo">
    <Property Type="Int32" Name="EnrollmentID" Nullable="false" />
    <Property Type="Decimal" Name="Grade" Nullable="true"
              Precision="3" Scale="2" />
    <Property Type="Int32" Name="CourseID" Nullable="false" />
    <Property Type="Int32" Name="StudentID" Nullable="false" />
</ComplexType>
```

为了创建返回以前复杂类型的实例的函数导入，必须在**ComplexTypeMapping**元素中定义存储过程返回的列与实体类型返回的列之间的映射：

```
<FunctionImportMapping FunctionImportName="GetGrades"
                       FunctionName="SchoolModel.Store.GetGrades" >
    <ResultMapping>
        <ComplexTypeMapping TypeName="SchoolModel.GradeInfo">
            <ScalarProperty Name="EnrollmentID" ColumnName="enroll_id"/>
            <ScalarProperty Name="CourseID" ColumnName="course_id"/>
            <ScalarProperty Name="StudentID" ColumnName="student_id"/>
            <ScalarProperty Name="Grade" ColumnName="grade"/>
        </ComplexTypeMapping>
    </ResultMapping>
</FunctionImportMapping>
```

Condition 元素 (MSL)

映射规范语言 (MSL) 中的**Condition**元素对概念模型和基础数据库之间的映射施加条件。如果满足子**Condition**元素中指定的所有条件，则在 XML 节点内定义的映射将有效。否则，该映射无效。例如，如果 **MappingFragment** 元素包含一个或多个**Condition**子元素，则在满足子**Condition**元素的所有条件时，在**MappingFragment**节点内定义的映射将有效。

每个条件可以应用于名称 (概念模型实体属性的名称，由**Name**属性指定)，也可以应用于**columnname** (数据库中列的名称，由**ColumnName**特性指定)。设置**Name**属性时，将根据实体属性值检查条件。设置**ColumnName**属性后，将根据列值检查条件。只能在**Condition**元素中指定**Name**或**ColumnName**特性中的一个。

NOTE

如果在 FunctionImportMapping 元素中使用Condition元素，则仅Name属性不适用。

Condition元素可以是以下元素的子元素：

- AssociationSetMapping
- ComplexProperty
- EntitySetMapping
- MappingFragment
- EntityTypeMapping

Condition元素不能有子元素。

适用的属性

下表描述了适用于Condition元素的属性：

属性	说明	示例
ColumnName	是 表列的名称，其值用于计算条件。	
IsNull	是 True 或 False。如果值为True，列值为null，或者如果值为False且列值不为null，则条件为 true。否则，条件为 False。 不能同时使用IsNull和Value属性。	
■	是 要与列值进行比较的值。如果值不同，则该条件为 True。否则，条件为 False。 不能同时使用IsNull和Value属性。	
■	是 概念模型实体属性的名称，其值用于计算条件。 如果在 FunctionImportMapping 元素中使用Condition元素，则此属性不适用。	

示例

下面的示例将条件元素显示为MappingFragment元素的子元素。当 "雇佣日期" 不为 Null 并且EnrollmentDate为 null 时，将在 SchoolModel类型与Person表的PersonID和雇用日期列之间映射数据。如果EnrollmentDate不为 Null 且雇佣日期为 null，则会在SchoolModel类型与Person表的PersonID和注册列之间映射数据。

```

<EntitySetMapping Name="People">
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Person)">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="LastName" ColumnName="LastName" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Instructor)">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="HireDate" ColumnName="HireDate" />
      <Condition ColumnName="HireDate" IsNull="false" />
      <Condition ColumnName="EnrollmentDate" IsNull="true" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Student)">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="EnrollmentDate"
                     ColumnName="EnrollmentDate" />
      <Condition ColumnName="EnrollmentDate" IsNull="false" />
      <Condition ColumnName="HireDate" IsNull="true" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>

```

DeleteFunction 元素 (MSL)

映射规范语言 (MSL) 中的 **DeleteFunction** 元素将概念模型中的实体类型或关联的删除函数映射到基础数据库中的存储过程。将修改函数映射到其中的存储过程必须在存储模型中声明。有关详细信息，请参阅 [Function 元素 \(SSDL\)](#)。

NOTE

如果没有将实体类型的所有三个插入、更新或删除操作映射到存储过程，则在运行时执行并引发 `UpdateException` 时，未映射的操作将失败。

应用于 EntityTypeMapping 的 DeleteFunction

当应用于 **EntityTypeMapping** 元素时，**DeleteFunction** 元素会将概念模型中的实体类型的删除函数映射到存储过程。

DeleteFunction 元素在应用于 **EntityTypeMapping** 元素时可以具有以下子元素：

- `AssociationEnd` (零个或多个)
- `ComplexProperty` (零个或多个)
- `ScalarProperty` (零个或多个)

适用的属性

下表描述了可应用于 **EntityTypeMapping** 元素的 **DeleteFunction** 元素的特性。

属性	说明	示例
<code>FunctionName</code>	是	删除函数要映射到的存储过程的命名空间限定名称。存储过程必须在存储模型中声明。
<code>RowsAffectedParameter</code>	是	返回受影响行数的输出参数的名称。

示例

下面的示例基于 School 模型，并显示将 Person 实体类型的 delete 函数映射到 DeletePerson 存储过程的 DeleteFunction 元素。DeletePerson 存储过程在存储模型中声明。

```
<EntitySetName Name="People">
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="LastName" ColumnName="LastName" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="HireDate" ColumnName="HireDate" />
      <ScalarProperty Name="EnrollmentDate"
        ColumnName="EnrollmentDate" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <ModificationFunctionMapping>
      <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName" />
        <ScalarProperty Name="LastName" ParameterName="LastName" />
        <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
      </InsertFunction>
      <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate"
          Version="Current" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate"
          Version="Current" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName"
          Version="Current" />
        <ScalarProperty Name="LastName" ParameterName="LastName"
          Version="Current" />
        <ScalarProperty Name="PersonID" ParameterName="PersonID"
          Version="Current" />
      </UpdateFunction>
      <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
        <ScalarProperty Name="PersonID" ParameterName="PersonID" />
      </DeleteFunction>
    </ModificationFunctionMapping>
  </EntityTypeMapping>
</EntitySetName>
```

应用于 AssociationSetMapping 的 DeleteFunction

当应用于 AssociationSetMapping 元素时，DeleteFunction 元素会将概念模型中的关联的删除函数映射到存储过程。

DeleteFunction 元素在应用于 AssociationSetMapping 元素时可以具有以下子元素：

- EndProperty

适用的属性

下表描述了可应用于 AssociationSetMapping 元素的 DeleteFunction 元素的特性。

属性	说明	示例
FunctionName	是	删除函数要映射到的存储过程的命名空间限定名称。存储过程必须在存储模型中声明。

RowsAffectedParameter	是	返回受影响行数的输出参数的名称。

示例

下面的示例基于 School 模型，并显示了用于将courseinstructor.courseid关联的删除函数映射到DeleteCourseInstructor存储过程的DeleteFunction元素。DeleteCourseInstructor存储过程在存储模型中声明。

```
<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>
```

EndProperty 元素 (MSL)

映射规范语言 (MSL) 中的EndProperty元素定义概念模型关联和基础数据库的结束或修改函数之间的映射。属性-列映射在子 ScalarProperty 元素中指定。

当使用EndProperty元素定义概念模型关联末尾的映射时，它是 AssociationSetMapping 元素的子元素。当使用EndProperty元素定义概念模型关联的修改函数的映射时，它是 InsertFunction 元素或 DeleteFunction 元素的子元素。

EndProperty元素可以具有以下子元素：

- ScalarProperty (零个或多个)

适用的属性

下表描述了适用于EndProperty元素的属性：

名称	是	要映射的关联端的名称。

示例

下面的示例显示一个AssociationSetMapping元素，在该元素中，概念模型中的FK_课程_部门关联映射到数据库中的课程表。在子EndProperty元素中指定关联类型属性与表列之间的映射。

```
<AssociationSetMapping Name="FK_Course_Department"
    TypeName="SchoolModel.FK_Course_Department"
    StoreEntitySet="Course">
    <EndProperty Name="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
</AssociationSetMapping>
```

示例

下面的示例显示了EndProperty元素，该元素将关联(courseinstructor.courseid)的插入和删除函数映射到基础数据库中的存储过程。映射到的函数在存储模型中声明。

```
<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>
```

EntityContainerMapping 元素 (MSL)

映射规范语言(MSL)中的EntityContainerMapping元素将概念模型中的实体容器映射到存储模型中的实体容器。EntityContainerMapping元素是 Mapping 元素的子元素。

EntityContainerMapping元素可以具有以下子元素(按所列顺序)：

- EntitySetMapping(零个或多个)
- AssociationSetMapping (零个或多个)
- FunctionImportMapping (零个或多个)

适用的属性

下表介绍可应用于EntityContainerMapping元素的特性。

属性	说明	示例
StorageModelContainer	是	要映射到的存储模型实体容器的名称。
CdmEntityContainer	是	要映射的概念模型实体容器的名称。
GenerateUpdateViews	是	True 或 False。如果 False，则不生成任何更新视图。如果你具有只读映射，则此属性应设置为False，因为数据可能无法成功往返。 默认值为 True。

示例

下面的示例演示了一个EntityContainerMapping元素，该元素将SchoolModelEntities容器(概念模型实体容器)映射到SchoolModelStoreContainer容器(存储模型实体容器)：

```
<EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
        <MappingFragment StoreEntitySet="Course">
            <ScalarProperty Name="CourseID" ColumnName="CourseID" />
            <ScalarProperty Name="Title" ColumnName="Title" />
            <ScalarProperty Name="Credits" ColumnName="Credits" />
            <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="c.Department">
        <MappingFragment StoreEntitySet="Department">
            <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
            <ScalarProperty Name="Name" ColumnName="Name" />
            <ScalarProperty Name="Budget" ColumnName="Budget" />
            <ScalarProperty Name="StartDate" ColumnName="StartDate" />
            <ScalarProperty Name="Administrator" ColumnName="Administrator" />
        </MappingFragment>
    </EntityTypeMapping>
    </EntityTypeMapping>
</EntityContainerMapping>
```

EntityTypeMapping 元素 (MSL)

映射规范语言 (MSL) 中的EntityTypeMapping元素将概念模型实体集中的所有类型都映射到存储模型中的实体集。概念模型中的实体集为同一类型(和派生类型)实体的实例的逻辑容器。存储模型中的实体集表示基础数据库中的一个表或视图。概念模型实体集由EntityTypeMapping元素的Name特性的值指定。映射到的表或视图由每个子MappingFragment元素或EntityTypeMapping元素本身中的StoreEntitySet属性指定。

EntityTypeMapping元素可以具有以下子元素：

- EntityTypeMapping(零个或多个)
- QueryView (零个或一个)
- MappingFragment (零个或多个)

适用的属性

下表介绍可应用于EntityTypeMapping元素的特性。

属性	说明	示例
■	是	要映射的概念模型实体集的名称。
TypeName 1	是	要映射的概念模型实体类型的名称。
StoreEntitySet 1	是	要映射到的存储模型实体集的名称。
MakeColumnsDistinct	是	True 或 False ，具体取决于是否只返回不同的行。 如果此特性设置为 True ，则 EntityTypeMapping 元素的 GenerateUpdateViews 特性必须设置为 False 。

1可以使用**TypeName**和**StoreEntitySet**特性替代 **EntityTypeMapping** 和 **MappingFragment** 子元素，以便将单个实体类型映射到单个表。

示例

下面的示例显示一个**EntitySetMapping**元素，该元素将概念模型的课程实体集中的三个类型(基类型和两个派生类型)映射到基础数据库中的三个不同表。这些表由每个**MappingFragment**元素中的**StoreEntitySet**属性指定。

```
<EntitySetMapping Name="Courses">
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.Course)">
    <MappingFragment StoreEntitySet="Course">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      <ScalarProperty Name="Credits" ColumnName="Credits" />
      <ScalarProperty Name="Title" ColumnName="Title" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.OnlineCourse)">
    <MappingFragment StoreEntitySet="OnlineCourse">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="URL" ColumnName="URL" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel1.OnsiteCourse)">
    <MappingFragment StoreEntitySet="OnsiteCourse">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="Time" ColumnName="Time" />
      <ScalarProperty Name="Days" ColumnName="Days" />
      <ScalarProperty Name="Location" ColumnName="Location" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

EntityTypeMapping 元素 (MSL)

映射规范语言 (MSL) 中的**EntityTypeMapping**元素定义概念模型中的实体类型与基础数据库中的表或视图之间的映射。有关概念模型实体类型与基础数据库表或视图的信息，请参见 **EntityType** 元素 (CSDL) 和 **EntitySet** 元素 (SSDL)。要映射的概念模型实体类型由**EntityTypeMapping**元素的**TypeName**特性指定。要映射的表或视图由子**MappingFragment**元素的**StoreEntitySet**属性指定。

ModificationFunctionMapping 子元素可以用于将实体类型的插入、更新、或删除函数映射到数据库中的存储过程。

EntityTypeMapping元素可以具有以下子元素：

- MappingFragment (零个或多个)
- ModificationFunctionMapping (零个或一个)
- ScalarProperty
- 条件

NOTE

MappingFragment和ModificationFunctionMapping元素不能同时是EntityTypeMapping元素的子元素。

NOTE

当ScalarProperty和Condition元素在 FunctionImportMapping 元素中使用时, 它们只能是EntityTypeMapping元素的子元素。

适用的属性

下表介绍可应用于EntityTypeMapping元素的特性。

属性	说明	示例
TypeName	是 要映射的概念模型实体类型的命名空间限定的名称。 如果类型为抽象类型或派生类型, 则值必须为 <code>IsOfType(Namespace-qualified_type_name)</code> 。	

示例

下面的示例演示一个具有两个子EntityTypeMapping元素的 EntitySetMapping 元素。在第一个EntityTypeMapping元素中, SchoolModel实体类型映射到person表。在第二个EntityTypeMapping元素中, SchoolModel类型的更新功能映射到数据库中的存储过程UpdatePerson。

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <ScalarProperty Name="EnrollmentDate" ColumnName="EnrollmentDate" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="SchoolModel.Person">
        <ModificationFunctionMapping>
            <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
                <ScalarProperty Name="EnrollmentDate" ParameterName="EnrollmentDate"
                               Version="Current" />
                <ScalarProperty Name="HireDate" ParameterName="HireDate"
                               Version="Current" />
                <ScalarProperty Name="FirstName" ParameterName="FirstName"
                               Version="Current" />
                <ScalarProperty Name="LastName" ParameterName="LastName"
                               Version="Current" />
                <ScalarProperty Name="PersonID" ParameterName="PersonID"
                               Version="Current" />
            </UpdateFunction>
        </ModificationFunctionMapping>
    </EntityTypeMapping>
</EntitySetMapping>

```

示例

下一示例演示其中根类型为抽象类型的类型层次结构的映射。请注意，将 `IsOfType` 语法用于 `TypeName` 特性。

```

<EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Person)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Instructor)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="HireDate" ColumnName="HireDate" />
            <Condition ColumnName="HireDate" IsNull="false" />
            <Condition ColumnName="EnrollmentDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.Student)">
        <MappingFragment StoreEntitySet="Person">
            <ScalarProperty Name="PersonID" ColumnName="PersonID" />
            <ScalarProperty Name="EnrollmentDate"
                           ColumnName="EnrollmentDate" />
            <Condition ColumnName="EnrollmentDate" IsNull="false" />
            <Condition ColumnName="HireDate" IsNull="true" />
        </MappingFragment>
    </EntityTypeMapping>
</EntitySetMapping>

```

FunctionImportMapping 元素 (MSL)

映射规范语言 (MSL) 中的 `FunctionImportMapping` 元素定义概念模型中的函数导入与基础数据库中的存储过程或函数之间的映射。函数导入必须在概念模型中进行声明，存储的过程必须在存储模型中进行声明。有关详细信

息，请参阅 [FunctionImport 元素\(CSDL\)](#) 和 [Function 元素\(SSDL\)](#)。

NOTE

默认情况下，如果函数导入返回概念模型实体类型或复杂类型，则基础存储过程返回的列名称必须与概念模型类型中的属性名称完全匹配。如果列名称与属性名称不完全匹配，则必须在 [ResultMapping](#) 元素中定义映射。

[FunctionImportMapping](#) 元素可以具有以下子元素：

- [ResultMapping](#) (零个或多个)

适用的属性

下表描述了适用于 [FunctionImportMapping](#) 元素的属性：

属性	说明	示例
FunctionImportName	是	概念模型中要映射的函数导入的名称。
FunctionName	是	存储模型中要映射的函数的命名空间限定名称。

示例

下面的示例基于 School 模型。请考虑在存储模型中使用以下函数：

```
<Function Name="GetStudentGrades" Aggregate="false"
          BuiltIn="false" NiladicFunction="false"
          IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
          Schema="dbo">
    <Parameter Name="StudentID" Type="int" Mode="In" />
</Function>
```

另请考虑在概念模型中使用此函数导入：

```
<FunctionImport Name="GetStudentGrades" EntitySet="StudentGrades"
                ReturnType="Collection(SchoolModel.StudentGrade)">
    <Parameter Name="StudentID" Mode="In" Type="Int32" />
</FunctionImport>
```

下面的示例显示一个 [FunctionImportMapping](#) 元素，该元素用于将上述函数和函数导入映射到彼此：

```
<FunctionImportMapping FunctionImportName="GetStudentGrades"
                           FunctionName="SchoolModel.Store.GetStudentGrades" />
```

InsertFunction 元素 (MSL)

映射规范语言 (MSL) 中的 [InsertFunction](#) 元素将概念模型中的实体类型或关联的插入函数映射到基础数据库中的存储过程。将修改函数映射到其中的存储过程必须在存储模型中声明。有关详细信息，请参阅 [Function 元素 \(SSDL\)](#)。

NOTE

如果没有将实体类型的所有三个插入、更新或删除操作映射到存储过程，则在运行时执行并引发 `UpdateException` 时，未映射的操作将失败。

`InsertFunction` 元素可以是 `ModificationFunctionMapping` 元素的子元素，并应用于 `EntityTypeMapping` 元素或 `AssociationSetMapping` 元素。

应用于 `EntityTypeMapping` 的 `InsertFunction`

当应用于 `EntityTypeMapping` 元素时，`InsertFunction` 元素会将概念模型中的实体类型的插入函数映射到存储过程。

`InsertFunction` 元素在应用于 `EntityTypeMapping` 元素时可以具有以下子元素：

- `AssociationEnd` (零个或多个)
- `ComplexProperty` (零个或多个)
- `ResultBinding` (零个或一个)
- `ScalarProperty` (零个或多个)

适用的属性

下表描述了应用于 `EntityTypeMapping` 元素时可应用于 `InsertFunction` 元素的特性。

属性	说明	示例
<code>FunctionName</code>	是	插入函数要映射到的存储过程的命名空间限定名称。存储过程必须在存储模型中声明。
<code>RowsAffectedParameter</code>	是	返回受影响行数的输出参数的名称。

示例

下面的示例基于 `School` 模型，并显示了用于将 `Person` 实体类型的插入函数映射到 `InsertPerson` 存储过程的 `InsertFunction` 元素。`InsertPerson` 存储过程在存储模型中声明。

```

<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>

```

应用于 AssociationSetMapping 的 InsertFunction

当应用于 AssociationSetMapping 元素时，InsertFunction 元素会将概念模型中的关联的插入函数映射到存储过程。

InsertFunction 元素在应用于 AssociationSetMapping 元素时可以具有以下子元素：

- EndProperty

适用的属性

下表描述了可应用于 AssociationSetMapping 元素的 InsertFunction 元素的特性。

属性	说明	示例
FunctionName	是 插入函数要映射到的存储过程的命名空间限定名称。存储过程必须在存储模型中声明。	
RowsAffectedParameter	是 返回受影响行数的输出参数的名称。	

示例

下面的示例基于 School 模型，并显示了用于将 courseinstructor.courseid 关联的插入函数映射到 InsertCourseInstructor 存储过程的 InsertFunction 元素。InsertCourseInstructor 存储过程在存储模型中声明。

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

Mapping 元素 (MSL)

映射规范语言 (MSL) 中的 **mapping** 元素包含用于将概念模型中定义的对象映射到数据库的信息 (如存储模型中所述)。有关详细信息, 请参阅 CSDL 规范和 SSDL 规范。

Mapping 元素是映射规范的根元素。<https://schemas.microsoft.com/ado/2009/11/mapping/cs> 映射规范的 XML 命名空间。

映射元素可以具有以下子元素(按所列顺序) :

- 别名 (0个或多个)
- EntityContainerMapping (恰好一个)

MSL 中引用的概念模型类型和存储模型类型的名称必须由其相应的命名空间名称限定。有关概念模型命名空间名称的信息, 请参阅 Schema 元素 (CSDL)。有关存储模型命名空间名称的信息, 请参阅 Schema 元素 (SSDL)。可以使用 Alias 元素定义 MSL 中使用的命名空间的别名。

适用的属性

下表描述了可应用到 **Mapping** 元素的特性。

属性	说明	示例
空格键	是	C-S。这是固定值, 因此不能更改。

示例

下面的示例演示一个基于 School 模型的一部分的映射元素。有关 School 模型的详细信息, 请参阅快速入门 (实体框架) :

```

<Mapping Space="C-S"
  xmlns="https://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <Alias Key="c" Value="SchoolModel"/>
  <EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolModelEntities">
    <EntityTypeMapping TypeName="c.Course">
      <MappingFragment StoreEntitySet="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
        <ScalarProperty Name="Title" ColumnName="Title" />
        <ScalarProperty Name="Credits" ColumnName="Credits" />
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  <EntitySetMapping Name="Departments">
    <EntityTypeMapping TypeName="c.Department">
      <MappingFragment StoreEntitySet="Department">
        <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
        <ScalarProperty Name="Name" ColumnName="Name" />
        <ScalarProperty Name="Budget" ColumnName="Budget" />
        <ScalarProperty Name="StartDate" ColumnName="StartDate" />
        <ScalarProperty Name="Administrator" ColumnName="Administrator" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  </EntityContainerMapping>
</Mapping>

```

MappingFragment 元素 (MSL)

映射规范语言 (MSL) 中的 **MappingFragment** 元素定义概念模型实体类型的属性与数据库中的表或视图之间的映射。有关概念模型实体类型与基础数据库表或视图的信息，请参见 **EntityType** 元素 (CSDL) 和 **EntitySet** 元素 (SSDL)。**MappingFragment** 可以是 **EntityTypeMapping** 元素或 **EntitySetMapping** 元素的子元素。

MappingFragment 元素可以具有以下子元素：

- **ComplexType** (零个或多个)
- **ScalarProperty** (零个或多个)
- 条件 (零个或多个)

适用的属性

下表介绍可应用于 **MappingFragment** 元素的特性。

属性	说明	示例
StoreEntitySet	是	要映射的表或视图的名称。
MakeColumnsDistinct	是	True 或 False ，具体取决于是否只返回不同的行。 如果此特性设置为 True ，则 EntityContainerMapping 元素的 GenerateUpdateViews 特性必须设置为 False 。

示例

下面的示例演示一个 **MappingFragment** 元素作为 **EntityTypeMapping** 元素的子元素。在此示例中，概念模型中的课程类型的属性将映射到数据库中课程表的列。

```
<EntitySetMapping Name="Courses">
  <EntityTypeMapping TypeName="SchoolModel.Course">
    <MappingFragment StoreEntitySet="Course">
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
      <ScalarProperty Name="Title" ColumnName="Title" />
      <ScalarProperty Name="Credits" ColumnName="Credits" />
      <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

示例

下面的示例演示一个**MappingFragment**元素作为**EntitySetMapping**元素的子元素。如上面的示例所示，概念模型中的课程类型的属性将映射到数据库中课程表的列。

```
<EntitySetMapping Name="Courses" TypeName="SchoolModel.Course">
  <MappingFragment StoreEntitySet="Course">
    <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    <ScalarProperty Name="Title" ColumnName="Title" />
    <ScalarProperty Name="Credits" ColumnName="Credits" />
    <ScalarProperty Name="DepartmentID" ColumnName="DepartmentID" />
  </MappingFragment>
</EntitySetMapping>
```

ModificationFunctionMapping 元素 (MSL)

映射规范语言 (MSL) 中的**ModificationFunctionMapping**元素将概念模型实体类型的插入、更新和删除函数映射到基础数据库中的存储过程。**ModificationFunctionMapping**元素还可以将概念模型中的多对多关联的插入和删除函数映射到基础数据库中的存储过程。将修改函数映射到其中的存储过程必须在存储模型中声明。有关详细信息，请参阅 [Function 元素 \(SSDL\)](#)。

NOTE

如果没有将实体类型的所有三个插入、更新或删除操作映射到存储过程，则在运行时执行并引发 `UpdateException` 时，未映射的操作将失败。

NOTE

如果将继承层次结构中的一个实体的修改函数映射到存储过程，则必须将该层次结构中所有类型的修改函数都映射到存储过程。

ModificationFunctionMapping元素可以是 **EntityTypeMapping** 元素或 **AssociationSetMapping** 元素的子元素。

ModificationFunctionMapping元素可以具有以下子元素：

- `DeleteFunction` (零个或一个)
- `InsertFunction` (零个或一个)
- `UpdateFunction` (零个或一个)

没有适用于**ModificationFunctionMapping**元素的特性。

示例

下面的示例演示了 School 模型中 "人员" 实体集的实体集映射。除了 `person` 实体类型的列映射外，还会显示 "人员" 类型的插入、更新和删除函数的映射。映射到的函数在存储模型中声明。

```

<EntitySetMapping Name="People">
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="LastName" ColumnName="LastName" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="HireDate" ColumnName="HireDate" />
      <ScalarProperty Name="EnrollmentDate"
        ColumnName="EnrollmentDate" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <ModificationFunctionMapping>
      <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName" />
        <ScalarProperty Name="LastName" ParameterName="LastName" />
        <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
      </InsertFunction>
      <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate"
          Version="Current" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate"
          Version="Current" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName"
          Version="Current" />
        <ScalarProperty Name="LastName" ParameterName="LastName"
          Version="Current" />
        <ScalarProperty Name="PersonID" ParameterName="PersonID"
          Version="Current" />
      </UpdateFunction>
      <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
        <ScalarProperty Name="PersonID" ParameterName="PersonID" />
      </DeleteFunction>
    </ModificationFunctionMapping>
  </EntityTypeMapping>
</EntitySetMapping>

```

示例

下面的示例演示了 School 模型中courseinstructor.courseid关联集的关联集映射。除了courseinstructor.courseid关联的列映射外，还会显示courseinstructor.courseid关联的插入和删除函数的映射。映射到的函数在存储模型中声明。

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

QueryView 元素 (MSL)

映射规范语言 (MSL) 中的 **QueryView** 元素定义概念模型中的实体类型或关联与基础数据库中的表之间的只读映射。映射是使用针对存储模型计算的实体 SQL 查询定义的，你可以根据概念模型中的实体或关联来表示结果集。因为查询视图是只读的，所以不能使用标准更新命令来更新查询视图所定义的类型。可以使用修改函数来更新这些类型。有关详细信息，请参阅如何：将修改函数映射到存储过程。

NOTE

在 **QueryView** 元素中，不支持实体 SQL 包含 **GroupBy**、组聚合或导航属性的表达式。

QueryView 元素可以是 **EntityTypeMapping** 元素或 **AssociationSetMapping** 元素的子元素。在前一种情况下，查询视图定义概念模型中实体的只读映射。在后一种情况下，查询视图定义概念模型中关联的只读映射。

NOTE

如果 **AssociationSetMapping** 元素用于与引用约束的关联，则忽略 **AssociationSetMapping** 元素。有关详细信息，请参阅 **ReferentialConstraint** 元素 (CSDL)。

QueryView 元素不能具有任何子元素。

适用的属性

下表介绍可应用于 **QueryView** 元素的特性。

TypeName	是	要由查询视图映射的概念模型类型的名称。

示例

下面的示例将QueryView元素显示为EntitySetMapping元素的子元素，并为 School 模型中的部门实体类型定义查询视图映射。

```
<EntitySetMapping Name="Departments" >
  <QueryView>
    SELECT VALUE SchoolModel.Department(d.DepartmentID,
                                         d.Name,
                                         d.Budget,
                                         d.StartDate)
    FROM SchoolModelStoreContainer.Department AS d
    WHERE d.Budget > 150000
  </QueryView>
</EntitySetMapping>
```

由于该查询仅返回存储模型中 "部门" 类型的成员子集，因此，School 模型中的部门类型已根据以下映射进行了修改：

```
<EntityType Name="Department">
  <Key>
    <PropertyRef Name="DepartmentID" />
  </Key>
  <Property Type="Int32" Name="DepartmentID" Nullable="false" />
  <Property Type="String" Name="Name" Nullable="false"
            MaxLength="50" FixedLength="false" Unicode="true" />
  <Property Type="Decimal" Name="Budget" Nullable="false"
            Precision="19" Scale="4" />
  <Property Type="DateTime" Name="StartDate" Nullable="false" />
  <NavigationProperty Name="Courses"
    Relationship="SchoolModel.FK_Course_Department"
    FromRole="Department" ToRole="Course" />
</EntityType>
```

示例

下一个示例显示了QueryView元素作为AssociationSetMapping元素的子元素，并为 School 模型中的 `FK_Course_Department` 关联定义了只读映射。

```

<EntityContainerMapping StorageEntityContainer="SchoolModelStoreContainer"
    CdmEntityContainer="SchoolEntities">
    <EntitySetMapping Name="Courses" >
        <QueryView>
            SELECT VALUE SchoolModel.Course(c.CourseID,
                c.Title,
                c.Credits)
            FROM SchoolModelStoreContainer.Course AS c
        </QueryView>
    </EntitySetMapping>
    <EntitySetMapping Name="Departments" >
        <QueryView>
            SELECT VALUE SchoolModel.Department(d.DepartmentID,
                d.Name,
                d.Budget,
                d.StartDate)
            FROM SchoolModelStoreContainer.Department AS d
            WHERE d.Budget > 150000
        </QueryView>
    </EntitySetMapping>
    <AssociationSetMapping Name="FK_Course_Department" >
        <QueryView>
            SELECT VALUE SchoolModel.FK_Course_Department(
                CREATEREF(SchoolEntities.Departments, row(c.DepartmentID), SchoolModel.Department),
                CREATEREF(SchoolEntities.Courses, row(c.CourseID)) )
            FROM SchoolModelStoreContainer.Course AS c
        </QueryView>
    </AssociationSetMapping>
</EntityContainerMapping>

```

Comments

可以定义查询视图来实现以下方案：

- 在概念模型中定义一个实体，该实体不包含存储模型中的实体的所有属性。这包括没有默认值并且不支持null值的属性。
- 将存储模型中计算的列映射到概念模型中实体类型的属性。
- 定义一个映射，其中用于对概念模型中的实体进行分区的条件不基于相等性。使用Condition元素指定条件映射时，所提供的条件必须等于指定的值。有关详细信息，请参阅 Condition 元素 (MSL)。
- 将存储模型中的同一列映射到概念模型中的多个类型。
- 将多个类型映射到同一个表。
- 在概念模型中定义不基于关系架构中的外键的关联。
- 使用自定义业务逻辑设置概念模型中的属性值。例如，您可以将数据源中的字符串值 "T" 映射到概念模型中的true值(布尔值)。
- 为查询结果定义条件筛选器。
- 强制对概念模型中的数据施加比存储模型中更少的限制。例如，您可以将概念模型中的属性设置为可以为 null，即使它映射到的列不支持null值。

为实体定义查询视图时需要考虑以下注意事项：

- 查询视图是只读的。只能使用修改函数来更新实体。
- 通过查询视图定义实体类型时，必须也通过查询视图来定义所有相关实体。
- 当您将多对多关联映射到存储模型中表示关系架构中的链接表的实体时，必须在此链接表的AssociationSetMapping元素中定义QueryView元素。
- 必须为类型层次结构中的所有类型定义查询视图。可以使用下列方式来实现：
 - 使用单个QueryView元素指定单个实体 SQL 查询，该查询返回层次结构中所有实体类型的联合。
 - 使用单个QueryView元素，该元素指定单个实体 SQL 查询，该查询使用 CASE 运算符根据特定条件返回

层次结构中的特定实体类型。

- 对于层次结构中的特定类型，使用额外的QueryView元素。在这种情况下，请使用QueryView元素的TypeName特性来指定每个视图的实体类型。
- 定义查询视图时，不能在EntitySetMapping元素上指定StorageSetName属性。
- 定义查询视图时，EntitySetMapping元素不能同时包含属性映射。

ResultBinding 元素 (MSL)

当实体类型修改函数映射到基础数据库中的存储过程时，映射规范语言 (MSL) 中的ResultBinding元素将存储过程返回的列值映射到概念模型中的实体属性。例如，当 insert 存储过程返回标识列的值时，ResultBinding元素会将返回的值映射到概念模型中的实体类型属性。

ResultBinding元素可以是 InsertFunction 元素或 UpdateFunction 元素的子元素。

ResultBinding元素不能具有任何子元素。

适用的属性

下表描述了适用于ResultBinding元素的属性：

属性	说明	示例
EntityType	是	概念模型中要映射的实体属性的名称。
ColumnName	是	要映射的列的名称。

示例

下面的示例基于 School 模型，并显示用于将Person实体类型的插入函数映射到InsertPerson存储过程的InsertFunction元素。(InsertPerson存储过程如下所示，在存储模型中声明。)ResultBinding元素用于将存储过程(NewPersonID)返回的列值映射到实体类型属性(PersonID)。

```
<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>
```

以下 Transact-sql 介绍了**InsertPerson**存储过程：

```
CREATE PROCEDURE [dbo].[InsertPerson]
    @LastName nvarchar(50),
    @FirstName nvarchar(50),
    @HireDate datetime,
    @EnrollmentDate datetime
AS
    INSERT INTO dbo.Person (LastName,
                           FirstName,
                           HireDate,
                           EnrollmentDate)
    VALUES (@LastName,
            @FirstName,
            @HireDate,
            @EnrollmentDate);
    SELECT SCOPE_IDENTITY() as NewPersonID;
```

ResultMapping 元素 (MSL)

映射规范语言 (MSL) 中的**ResultMapping**元素定义概念模型中的函数导入与基础数据库中的存储过程之间的映射 (如果满足以下条件)：

- 函数导入返回一个概念模型实体类型或复杂类型。
- 存储过程返回的列的名称与实体类型或复杂类型中的属性名称不完全匹配。

默认情况下，存储过程返回的列与实体类型或复杂类型中的属性之间的映射基于列名称和属性名称。如果列名称与属性名称不完全匹配，则必须使用**ResultMapping**元素来定义映射。有关默认映射的示例，请参阅 [FunctionImportMapping 元素 \(MSL\)](#)。

ResultMapping元素是 [FunctionImportMapping](#) 元素的子元素。

ResultMapping元素可以具有以下子元素：

- [EntityTypeMapping](#)(零个或多个)
- [ComplexTypeMapping](#)

没有适用于**ResultMapping**元素的特性。

示例

请思考以下存储过程：

```
CREATE PROCEDURE [dbo].[GetGrades]
    @student_Id int
AS
    SELECT EnrollmentID as enroll_id,
           Grade as grade,
           CourseID as course_id,
           StudentID as student_id
    FROM dbo.StudentGrade
    WHERE StudentID = @student_Id
```

另请考虑使用以下概念模型实体类型：

```

<EntityType Name="StudentGrade">
  <Key>
    <PropertyRef Name="EnrollmentID" />
  </Key>
  <Property Name="EnrollmentID" Type="Int32" Nullable="false"
    annotation:StoreGeneratedPattern="Identity" />
  <Property Name="CourseID" Type="Int32" Nullable="false" />
  <Property Name="StudentID" Type="Int32" Nullable="false" />
  <Property Name="Grade" Type="Decimal" Precision="3" Scale="2" />
</EntityType>

```

若要创建返回以前实体类型的实例的函数导入，必须在**ResultMapping**元素中定义存储过程返回的列与实体类型返回的列之间的映射：

```

<FunctionImportMapping FunctionImportName="GetGrades"
  FunctionName="SchoolModel.Store.GetGrades" >
  <ResultMapping>
    <EntityTypeMapping TypeName="SchoolModel.StudentGrade">
      <ScalarProperty Name="EnrollmentID" ColumnName="enroll_id"/>
      <ScalarProperty Name="CourseID" ColumnName="course_id"/>
      <ScalarProperty Name="StudentID" ColumnName="student_id"/>
      <ScalarProperty Name="Grade" ColumnName="grade"/>
    </EntityTypeMapping>
  </ResultMapping>
</FunctionImportMapping>

```

ScalarProperty 元素 (MSL)

映射规范语言 (MSL) 中的**ScalarProperty**元素将概念模型实体类型、复杂类型或关联的属性映射到基础数据库中的表列或存储过程参数。

NOTE

将修改函数映射到其中的存储过程必须在存储模型中声明。有关详细信息，请参阅 [Function 元素 \(SSDL\)](#)。

ScalarProperty元素可以是以下元素的子元素：

- **MappingFragment**
- **InsertFunction**
- **UpdateFunction**
- **DeleteFunction**
- **EndProperty**
- **ComplexProperty**
- **ResultMapping**

作为**MappingFragment**、**ComplexProperty**或**EndProperty**元素的子元素，**ScalarProperty**元素会将概念模型中的属性映射到数据库中的某一列。作为**InsertFunction**、**UpdateFunction**或**DeleteFunction**元素的子元素，**ScalarProperty**元素会将概念模型中的属性映射到存储过程参数。

ScalarProperty元素不能具有任何子元素。

适用的属性

应用于**ScalarProperty**元素的特性因元素的角色而异。

下表介绍了在**ScalarProperty**元素用于将概念模型属性映射到数据库中的列时适用的特性：

■	是	要映射的概念模型属性的名称。
ColumnName	是	要映射的表列的名称。

下表描述了当ScalarProperty元素用于将概念模型属性映射到存储过程参数时适用的特性：

■	是	要映射的概念模型属性的名称。
ParameterName	是	正在映射的参数的名称。
■	是	■或■值，具体取决于是否应将当前值或属性的原始值用于并发检查。

示例

下面的示例演示了两种方法中使用的ScalarProperty元素：

- 将**person**实体类型的属性映射到**person**表的列。
- 将**Person**实体类型的属性映射到**UpdatePerson**存储过程的参数。存储过程在存储模型中声明。

```

<EntitySetMapping Name="People">
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <MappingFragment StoreEntitySet="Person">
      <ScalarProperty Name="PersonID" ColumnName="PersonID" />
      <ScalarProperty Name="LastName" ColumnName="LastName" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="HireDate" ColumnName="HireDate" />
      <ScalarProperty Name="EnrollmentDate"
        ColumnName="EnrollmentDate" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="SchoolModel.Person">
    <ModificationFunctionMapping>
      <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName" />
        <ScalarProperty Name="LastName" ParameterName="LastName" />
        <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
      </InsertFunction>
      <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
        <ScalarProperty Name="EnrollmentDate"
          ParameterName="EnrollmentDate"
          Version="Current" />
        <ScalarProperty Name="HireDate" ParameterName="HireDate"
          Version="Current" />
        <ScalarProperty Name="FirstName" ParameterName="FirstName"
          Version="Current" />
        <ScalarProperty Name="LastName" ParameterName="LastName"
          Version="Current" />
        <ScalarProperty Name="PersonID" ParameterName="PersonID"
          Version="Current" />
      </UpdateFunction>
      <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
        <ScalarProperty Name="PersonID" ParameterName="PersonID" />
      </DeleteFunction>
    </ModificationFunctionMapping>
  </EntityTypeMapping>
</EntitySetMapping>

```

示例

下一个示例显示了**ScalarProperty**元素，该元素用于将概念模型关联的插入和删除函数映射到数据库中的存储过程。存储过程在存储模型中声明。

```

<AssociationSetMapping Name="CourseInstructor"
    TypeName="SchoolModel.CourseInstructor"
    StoreEntitySet="CourseInstructor">
    <EndProperty Name="Person">
        <ScalarProperty Name="PersonID" ColumnName="PersonID" />
    </EndProperty>
    <EndProperty Name="Course">
        <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </EndProperty>
    <ModificationFunctionMapping>
        <InsertFunction FunctionName="SchoolModel.Store.InsertCourseInstructor" >
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </InsertFunction>
        <DeleteFunction FunctionName="SchoolModel.Store.DeleteCourseInstructor">
            <EndProperty Name="Course">
                <ScalarProperty Name="CourseID" ParameterName="courseId"/>
            </EndProperty>
            <EndProperty Name="Person">
                <ScalarProperty Name="PersonID" ParameterName="instructorId"/>
            </EndProperty>
        </DeleteFunction>
    </ModificationFunctionMapping>
</AssociationSetMapping>

```

UpdateFunction 元素 (MSL)

映射规范语言 (MSL) 中的 **UpdateFunction** 元素将概念模型中的实体类型的更新函数映射到基础数据库中的存储过程。将修改函数映射到其中的存储过程必须在存储模型中声明。有关详细信息, 请参阅 **Function** 元素 (SSDL)。

NOTE

如果没有将实体类型的所有三个插入、更新或删除操作映射到存储过程, 则在运行时执行并引发 **UpdateException** 时, 未映射的操作将失败。

UpdateFunction 元素可以是 **ModificationFunctionMapping** 元素的子元素并应用于 **EntityTypeMapping** 元素。

UpdateFunction 元素可以具有以下子元素:

- **AssociationEnd** (零个或多个)
- **ComplexProperty**(零个或多个)
- **ResultBinding** (零个或一个)
- **ScalarProperty** (零个或多个)

适用的属性

下表介绍可应用于 **UpdateFunction** 元素的特性。

FunctionName	是	更新函数要映射到的存储过程的命名空间限定名称。存储过程必须在存储模型中声明。
RowsAffectedParameter	是	返回受影响行数的输出参数的名称。

示例

下面的示例基于 School 模型，并显示用于将 Person 实体类型的更新函数映射到 UpdatePerson 存储过程的 UpdateFunction 元素。UpdatePerson 存储过程在存储模型中声明。

```
<EntityTypeMapping TypeName="SchoolModel.Person">
  <ModificationFunctionMapping>
    <InsertFunction FunctionName="SchoolModel.Store.InsertPerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName" />
      <ScalarProperty Name="LastName" ParameterName="LastName" />
      <ResultBinding Name="PersonID" ColumnName="NewPersonID" />
    </InsertFunction>
    <UpdateFunction FunctionName="SchoolModel.Store.UpdatePerson">
      <ScalarProperty Name="EnrollmentDate"
                      ParameterName="EnrollmentDate"
                      Version="Current" />
      <ScalarProperty Name="HireDate" ParameterName="HireDate"
                      Version="Current" />
      <ScalarProperty Name="FirstName" ParameterName="FirstName"
                      Version="Current" />
      <ScalarProperty Name="LastName" ParameterName="LastName"
                      Version="Current" />
      <ScalarProperty Name="PersonID" ParameterName="PersonID"
                      Version="Current" />
    </UpdateFunction>
    <DeleteFunction FunctionName="SchoolModel.Store.DeletePerson">
      <ScalarProperty Name="PersonID" ParameterName="PersonID" />
    </DeleteFunction>
  </ModificationFunctionMapping>
</EntityTypeMapping>
```

SSDL 规范

2020/3/11 •

存储架构定义语言 (SSDL) 是一种基于 XML 的语言，用于描述实体框架应用程序的存储模型。

在实体框架应用程序中，存储模型元数据从 ssdl 文件（使用 SSDL 编写）加载到 StoreItemCollection 的实例中，可以使用中的方法进行访问。“System.object”类。实体框架使用存储模型元数据将针对概念模型的查询转换为特定于存储的命令。

Entity Framework Designer (EF 设计器) 在设计时将存储模型信息存储在 .edmx 文件中。在生成时，Entity Designer 使用 .edmx 文件中的信息来创建运行时实体框架所需的 ssdl 文件。

SSDL 的版本按 XML 命名空间进行区分。

SSDL 版本	XML 命名空间
SSDL v1	https://schemas.microsoft.com/ado/2006/04/edm/ssdl
SSDL v2	https://schemas.microsoft.com/ado/2009/02/edm/ssdl
SSDL v3	https://schemas.microsoft.com/ado/2009/11/edm/ssdl

Association 元素 (SSDL)

存储架构定义语言 (SSDL) 中的 Association 元素指定参与基础数据库中外键约束的表列。两个必需的子 End 元素指定位于关联两端的表和各端的重数。一个可选的 ReferentialConstraint 元素指定关联的主体端和依赖端以及参与列。如果不存在 ReferentialConstraint 元素，则必须使用 AssociationSetMapping 元素来指定关联的列映射。

Association 元素可以具有以下子元素（按所列顺序）：

- 文档（零个或一个）
- 结束（正好两个）
- ReferentialConstraint（零个或一个）
- 批注元素（零个或多个）

适用的属性

下表介绍可应用于 Association 元素的特性。

特性	值	说明
■	是	基础数据库中相应外键约束的名称。

NOTE

可以将任意数量的批注特性（自定义 XML 特性）应用于 Association 元素。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个 Association 元素，该元素使用 ReferentialConstraint 元素指定参与 FK_CustomerOrders 外键约束的列：

```

<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

AssociationSet 元素 (SSDL)

存储架构定义语言(SSDL)中的AssociationSet元素表示基础数据库中两个表之间的外键约束。参与外键约束的表列在 Association 元素中指定。与给定associationset元素对应的association元素是在AssociationSet元素的association特性中指定的。

SSDL 关联集通过 AssociationSetMapping 元素映射到 CSDL 关联集。但是，如果使用 ReferentialConstraint 元素定义给定 CSDL 关联集的 CSDL 关联，则不需要相应的AssociationSetMapping元素。在这种情况下，如果存在AssociationSetMapping元素，该元素定义的映射将被ReferentialConstraint元素重写。

AssociationSet元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个)
- End(零个或两个)
- 批注元素(零个或多个)

适用的属性

下表介绍可应用于AssociationSet元素的特性。

属性	说明	示例
■ Name	是	关联集表示的外键约束的名称。
■ Association	是	定义参与外键约束的列的关联的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于AssociationSet元素。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个AssociationSet元素，该元素表示基础数据库中的 `FK_CustomerOrders` foreign key 约束：

```

<AssociationSet Name="FK_CustomerOrders"
  Association="ExampleModel.Store.FK_CustomerOrders">
  <End Role="Customers" EntitySet="Customers" />
  <End Role="Orders" EntitySet="Orders" />
</AssociationSet>

```

CollectionType 元素 (SSDL)

存储架构定义语言(SSDL)中的**CollectionType**元素指定函数的返回类型是集合。**CollectionType**元素是**ReturnType**元素的子元素。集合的类型是通过使用**RowType**子元素指定的：

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**CollectionType**元素。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个函数，该函数使用**CollectionType**元素来指定该函数返回行的集合。

```
<Function Name="GetProducts" IsComposable="true" Schema="dbo">
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="ProductID" Type="int" Nullable="false" />
        <Property Name="CategoryID" Type="bigint" Nullable="false" />
        <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
        <Property Name="UnitPrice" Type="money" />
        <Property Name="Discontinued" Type="bit" />
      </RowType>
    </CollectionType>
  </ReturnType>
</Function>
```

CommandText 元素 (SSDL)

存储架构定义语言(SSDL)中的**CommandText**元素是 Function 元素的子元素，它允许您定义在数据库中执行的 SQL 语句。**CommandText**元素允许添加类似于数据库中的存储过程的功能，但在存储模型中定义**CommandText**元素。

CommandText元素不能有子元素。**CommandText**元素的正文必须是基础数据库的有效 SQL 语句。

无特性适用于**CommandText**元素。

示例

下面的示例演示具有子**CommandText**元素的函数元素。通过将**UpdateProductInOrder**函数导入到概念模型中，将其作为 **ObjectContext** 的方法公开。

```
<Function Name="UpdateProductInOrder" IsComposable="false">
  <CommandText>
    UPDATE Orders
    SET ProductId = @productId
    WHERE OrderId = @orderId;
  </CommandText>
  <Parameter Name="productId"
    Mode="In"
    Type="int"/>
  <Parameter Name="orderId"
    Mode="In"
    Type="int"/>
</Function>
```

DefiningQuery 元素 (SSDL)

使用存储架构定义语言(SSDL)中的DefiningQuery元素，可以直接在基础数据库中执行SQL语句。

DefiningQuery元素通常用于数据库视图，但该视图是在存储模型中定义的，而不是在数据库中定义的。

在DefiningQuery元素中定义的视图可以通过EntitySetMapping元素映射到概念模型中的实体类型。这些映射是只读的。

下面的SSDL语法显示了EntitySet的声明，后跟包含用于检索视图的查询的DefiningQuery元素。

```
<Schema>
  <EntitySet Name="Tables" EntityType="Self.STable">
    <DefiningQuery>
      SELECT TABLE_CATALOG,
        'test' as TABLE_SCHEMA,
        TABLE_NAME
      FROM INFORMATION_SCHEMA.TABLES
    </DefiningQuery>
  </EntitySet>
</Schema>
```

您可以使用实体框架中的存储过程来对视图启用读写方案。您可以使用数据源视图或实体SQL视图作为基表来检索数据和存储过程的更改处理。

您可以使用DefiningQuery元素将Microsoft SQL Server Compact为3.5。尽管SQL Server Compact 3.5不支持存储过程，但您可以通过DefiningQuery元素实现类似功能。另一个有用之处在于，可以创建存储过程以克服编程语言中使用的数据类型与数据源的数据类型的不匹配。您可以编写一个使用一组特定参数的DefiningQuery，然后使用一组不同的参数调用存储过程，例如，删除数据的存储过程。

Dependent元素(SSDL)

存储架构定义语言(SSDL)中的依赖元素是ReferentialConstraint元素的子元素，用于定义外键约束(也称为“引用约束”)的依赖端。Dependent元素指定引用主键列(或列)的表中的一个或多个列。PropertyRef元素指定要引用的列。主体元素指定依赖元素中指定的列所引用的主键列。

依赖元素可以具有以下子元素(按所列顺序)：

- PropertyRef(一个或多个)
- 批注元素(零个或多个)

适用的属性

下表介绍可应用于依赖元素的特性。

■	■	■
■	是	与相应的End元素的Role特性(如果使用)相同的值;否则为包含引用列的表的名称。

NOTE

可以将任意数量的批注特性(自定义XML特性)应用于■元素。然而，自定义特性可能不属于为CSDL保留的任何XML命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个Association元素，该元素使用ReferentialConstraint元素指定参与FK_CustomerOrders外键约束的列。Dependent元素将Order表的CustomerId列指定为约束的依赖端。

```

<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

Documentation 元素 (SSDL)

存储架构定义语言(SSDL)中的文档元素可用于提供有关在父元素中定义的对象的信息。

文档元素可以具有以下子元素(按所列顺序)：

- **摘要**: 父元素的简要说明。(零个或一个元素)
- **LongDescription**: 父元素的详细说明。(零个或一个元素)

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于文档元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例将文档元素显示为 EntityType 元素的子元素。

```

<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>

```

End 元素 (SSDL)

以存储架构定义语言(SSDL)表示的结束元素指定基础数据库中外键约束一端的表和行数。End元素可以是 Association 元素或 AssociationSet 元素的子元素。在每种情况下, 可能的子元素以及适用的特性都不同。

End 元素作为 Association 元素的子元素

End元素(作为Association元素的子元素)使用Type和多重性特性指定 foreign key 约束末尾的表和行数。外键约束的两端定义为 SSDL 关联的一部分;SSDL 关联必须仅具有两端。

结束元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个元素)
- OnDelete (零个或一个元素)

- 批注元素(零个或多个元素)

适用的属性

下表描述了当结束元素是Association元素的子元素时，可应用于该元素的特性。

属性	值	说明
类型	是	处于外键约束端的 SSDL 实体集的完全限定名。
■	是	相应 ReferentialConstraint 元素(如果使用)的主体或从属元素中的Role特性的值。
■	是	1、0或1或 *，具体取决于外键约束末尾的行数。 1 指示在外键约束端刚好存在一行。 0 .0 表示外键约束端处存在零个或一行。 * 指示外键约束端处存在零行、一行或多行。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个Association元素，该元素定义了FK_CustomerOrders外键约束。在每个End元素上指定的多重性值指示Orders表中的多个行可以与Customers表中的某一行相关联，但customers表中只有一个行可以与Orders表中的一行相关联。此外，OnDelete元素指示如果删除customers表中的行，则将删除Orders表中引用customers表中的特定行的所有行。

```
<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

End 元素作为 AssociationSet 元素的子元素

End元素(作为AssociationSet元素的子元素的子元素)指定基础数据库中外键约束一端的表。

结束元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个)
- 批注元素(零个或多个)

适用的属性

下表描述了当结束元素为AssociationSet元素的子元素时，可应用于该元素的特性。

属性	说明	示例
EntitySet	是	处于外键约束一端的 SSDL 实体集的名称。
■	是	在对应的 Association 元素的一个End元素上指定的一个■特性的值。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而，自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有AssociationSet元素的EntityContainer元素，其中包含两个End元素：

```
<EntityContainer Name="ExampleModelStoreContainer">
    <EntitySet Name="Customers"
        EntityType="ExampleModel.Store.Customers"
        Schema="dbo" />
    <EntitySet Name="Orders"
        EntityType="ExampleModel.Store.Orders"
        Schema="dbo" />
    <AssociationSet Name="FK_CustomerOrders"
        Association="ExampleModel.Store.FK_CustomerOrders">
        <End Role="Customers" EntitySet="Customers" />
        <End Role="Orders" EntitySet="Orders" />
    </AssociationSet>
</EntityContainer>
```

EntityContainer 元素 (SSDL)

存储架构定义语言(SSDL)中的EntityContainer元素描述实体框架应用程序中的基础数据源的结构：ssdl 实体集(在 EntitySet 元素中定义)表示数据库中的表，ssdl 实体类型(在 EntityType 元素中定义)表示表中的行，关联集(在 AssociationSet 元素中定义)表示数据库中的外键约束。存储模型实体容器通过 EntityContainerMapping 元素映射到概念模型实体容器。

EntityContainer元素可以有零个或一个文档元素。如果存在文档元素，则它必须位于所有其他子元素之前。

EntityContainer元素可包含零个或多个下列子元素(按所列顺序)：

- EntitySet
- AssociationSet
- 批注元素

适用的属性

下表介绍可应用于EntityContainer元素的特性。

属性	说明	示例
■	是	实体容器的名称。此名称不能包含句点(.)。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于EntityContainer元素。然而, 自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个EntityContainer元素, 该元素定义了两个实体集和一个关联集。注意, 实体类型和关联类型名称由概念模型命名空间名称限定。

```
<EntityContainer Name="ExampleModelStoreContainer">
  <EntitySet Name="Customers"
    EntityType="ExampleModel.Store.Customers"
    Schema="dbo" />
  <EntitySet Name="Orders"
    EntityType="ExampleModel.Store.Orders"
    Schema="dbo" />
  <AssociationSet Name="FK_CustomerOrders"
    Association="ExampleModel.Store.FK_CustomerOrders">
    <End Role="Customers" EntitySet="Customers" />
    <End Role="Orders" EntitySet="Orders" />
  </AssociationSet>
</EntityContainer>
```

EntitySet 元素 (SSDL)

存储架构定义语言(SSDL)中的EntitySet元素表示基础数据库中的表或视图。以 SSDL 表示的 EntityType 元素表示表或视图中的行。EntitySet元素的EntityType特性指定了表示 ssdl 实体集中的行的特定 SSDL 实体类型。CSDL 实体集和 SSDL 实体集之间的映射在 EntitySetMapping 元素中指定。

EntitySet元素可以具有以下子元素(按所列顺序):

- 文档(零个或一个元素)
- DefiningQuery (零个或一个元素)
- 批注元素

适用的属性

下表介绍可应用于EntitySet元素的特性。

NOTE

某些属性(此处未列出)可以用■别名进行限定。在更新模型时, 模型更新向导会使用这些特性。

■	■	■
■	是	实体集的名称。
EntityType	是	实体集包含其实例的实体类型的完全限定名称。
■	是	数据库架构。
■	是	数据库表。

NOTE

可以向**EntitySet**元素应用任意数量的批注特性(自定义 XML 特性)。然而, 自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有两个**EntitySet**元素和一个**AssociationSet**元素的**EntityContainer**元素:

```
<EntityContainer Name="ExampleModelStoreContainer">
  <EntitySet Name="Customers"
    EntityType="ExampleModel.Store.Customers"
    Schema="dbo" />
  <EntitySet Name="Orders"
    EntityType="ExampleModel.Store.Orders"
    Schema="dbo" />
  <AssociationSet Name="FK_CustomerOrders"
    Association="ExampleModel.Store.FK_CustomerOrders">
    <End Role="Customers" EntitySet="Customers" />
    <End Role="Orders" EntitySet="Orders" />
  </AssociationSet>
</EntityContainer>
```

EntityType 元素 (SSDL)

存储架构定义语言(SSDL)中的**EntityType**元素表示基础数据库的表或视图中的行。以 SSDL 表示的 EntitySet 元素表示行所在的表或视图。**EntityType**特性指定了表示 ssdl 实体集中的行的特定 SSDL 实体类型。SSDL 实体类型与 CSDL 实体类型之间的映射在 EntityTypeMapping 元素中指定。

EntityType元素可以具有以下子元素(按所列顺序):

- 文档(零个或一个元素)
- Key(零个或一个元素)
- 批注元素

适用的属性

下表介绍可应用于**EntityType**元素的特性。

属性	说明	示例
■	是	实体类型的名称。此值通常与以实体类型表示行的表的名称相同。此值可以不包含句点(.)。

NOTE

可以向**EntityType**元素应用任意数量的批注特性(自定义 XML 特性)。然而, 自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示具有两个属性的**EntityType**元素:

```

<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>

```

Function 元素 (SSDL)

存储架构定义语言(SSDL)中的**Function**元素指定基础数据库中存在的存储过程。

函数元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个)
- 参数(零个或多个)
- CommandText (0个或1个)
- ReturnType (零个或多个)
- 批注元素(零个或多个)

函数的返回类型必须与**returntype**元素或**returntype**属性(见下文)一起指定,但不能同时指定两者。

可以将在存储模型中指定的存储过程导入应用程序的概念模型。有关详细信息,请参阅[用存储过程查询](#)。**函数**元素还可用于在存储模型中定义自定义函数。

适用的属性

下表介绍可应用于**Function**元素的特性。

NOTE

某些属性(此处未列出)可以用■别名进行限定。在更新模型时,模型更新向导会使用这些特性。

属性	说明	示例
■	是	存储过程的名称。
ReturnType	是	存储过程的返回类型。
聚合	是	如果存储过程返回聚合值, ■ True ;否则■ False 。
L	是	如果函数是内置的 ¹ 函数, 则为 True ;否则为。否则■ False 。
StoreFunctionName	是	存储过程的名称。
NiladicFunction	是	如果函数是 niladic ² 函数, 则为 True ;否则■ False 。
IsComposable	是	如果函数是可组合的 ³ 函数, 则为 True ;否则■ False 。

ParameterTypeSemantics	是	定义用于解析函数重载的类型语义的枚举。该枚举是在提供程序清单中根据函数定义来定义的。默认值为AllowImplicitConversion。
■	是	在其中定义存储过程的架构的名称。

¹内置函数是在数据库中定义的函数。有关在存储模型中定义的函数的信息，请参阅 CommandText 元素(SSDL)。

² niladic 函数是不接受任何参数的函数，并且在调用时不需要括号。

³如果一个函数的输出可以是另一个函数的输入，则这两个函数是可组合的。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Function元素。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例显示了与UpdateOrderQuantity 存储过程对应的Function元素。该存储过程接受两个参数，且不返回值。

```
<Function Name="UpdateOrderQuantity"
    Aggregate="false"
    BuiltIn="false"
    NiladicFunction="false"
    IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
```

Key 元素 (SSDL)

以存储架构定义语言(SSDL)表示的Key元素表示基础数据库中的表的主键。Key是 EntityType 元素的子元素，它表示表中的行。通过引用EntityType元素中定义的一个或多个属性元素，在key元素中定义主键。

Key元素可以具有以下子元素(按所列顺序)：

- PropertyRef(一个或多个)
- 批注元素

没有任何特性适用于该键元素。

示例

下面的示例演示一个具有引用一个属性的键的EntityType元素：

```

<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>

```

OnDelete 元素 (SSDL)

存储架构定义语言(SSDL)中的**OnDelete**元素反映在删除参与 foreign key 约束的行时的数据库行为。如果将操作设置为**Cascade**, 则还将删除引用要删除的行的行。如果将操作设置为 "**无**", 则不会同时删除引用要删除行的行。**OnDelete**元素是结束元素的子元素。

OnDelete元素可以具有以下子元素(按所列顺序):

- 文档(零个或一个)
- 批注元素(零个或多个)

适用的属性

下表介绍可应用于**OnDelete**元素的特性。

属性	说明	示例
Action	是	Cascade 或 None 。(■值有效, 但行为与 None 相同。)

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于**OnDelete**元素。然而, 自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个**Association**元素, 该元素定义了**FK_CustomerOrders**外键约束。**OnDelete**元素指示在删除**customers**表中的某一行时, 该订单表中引用该特定行的所有行都将被删除。

```

<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>

```

Parameter 元素 (SSDL)

存储架构定义语言(SSDL)中的Parameter元素是Function元素的子元素，它指定数据库中存储过程的参数。

Parameter元素可以具有以下子元素(按所列顺序)：

- 文档(零个或一个)
- 批注元素(零个或多个)

适用的属性

下表描述了可应用于参数元素的特性。

属性	必需	说明
■	是	参数的名称。
类型	是	参数类型。
■	是	In、Out或InOut，具体取决于参数是输入、输出还是输入/输出参数。
MaxLength	是	参数的最大长度。
■	是	参数的精度。
■	是	参数的确定位数。
SRID	是	空间系统引用标识符。仅对空间类型的参数有效。有关详细信息，请参阅 SRID and SRID (SQL Server) 。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有两个参数元素的函数元素，这些参数元素指定输入参数：

```
<Function Name="UpdateOrderQuantity"
    Aggregate="false"
    BuiltIn="false"
    NiladicFunction="false"
    IsComposable="false"
    ParameterTypeSemantics="AllowImplicitConversion"
    Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
```

Principal 元素 (SSDL)

以存储架构定义语言(SSDL)表示的主体元素是ReferentialConstraint元素的子元素，用于定义外键约束(也称为“引用约束”)的主体端。主体元素指定表中一个或多个列(或列)引用的主键列。PropertyRef元素指定要引用的列。Dependent元素指定引用主体元素中指定的主键列的列。

主体元素可以具有以下子元素(按所列顺序)：

- PropertyRef(一个或多个)
- 批注元素(零个或多个)

适用的属性

下表介绍可应用于主体元素的特性。

■■■	■■■	■
■	是	与相应的 End 元素的Role特性(如果使用)相同的值;否则, 为包含被引用列的表的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于■元素。然而, 自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个 Association 元素, 该元素使用ReferentialConstraint元素指定参与FK_CustomerOrders外键约束的列。主体元素将Customer表的CustomerId列指定为约束的主体端。

```
<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

Property 元素 (SSDL)

存储架构定义语言(SSDL)中的属性元素表示基础数据库中的表的列。属性元素是 EntityType 元素的子元素, 表示表中的行。EntityType元素上定义的每个属性元素都表示一列。

属性元素不能具有任何子元素。

适用的属性

下表介绍可应用于属性元素的特性。

■■■	■■■	■
■	是	对应列的名称。
类型	是	对应列的类型。

属性	说明	示例
■ Null	是	True (默认值)或False , 具体取决于对应列是否可以具有 null 值。
■	是	对应列的默认值。
MaxLength	是	对应列的最大长度。
FixedLength	是	True或False , 具体取决于相应的列值是否将作为固定长度字符串存储。
■	是	对应列的精度。
■	是	对应列的小数位数。
Unicode	是	True或False , 具体取决于对应列值是否将存储为 Unicode 字符串。
■	是	指定要在数据源中使用的排序的字符串。
SRID	是	空间系统引用标识符。仅对空间类型的属性有效。有关详细信息, 请参阅 SRID 和 SRID (SQL Server) 。
StoreGeneratedPattern	是	■、■(如果相应的列值是在数据库中生成的标识)或■(如果在数据库中计算了相应的列值)。对于 RowType 属性无效。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于Property元素。然而, 自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个具有两个子属性元素的EntityType元素:

```
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

PropertyRef 元素 (SSDL)

存储架构定义语言(SSDL)中的PropertyRef元素引用 EntityType 元素上定义的属性, 以指示该属性将执行下列角色之一:

- 是EntityType表示的表的主键的一部分。一个或多个PropertyRef元素可用于定义主键。有关更多信息,请参见 Key 元素。
- 是引用约束的依赖端或主体端。有关更多信息,请参见 ReferentialConstraint 元素。

PropertyRef元素只能具有以下子元素:

- 文档(零个或一个)
- 批注元素

适用的属性

下表介绍可应用于PropertyRef元素的特性。

■	■	■
	■	所引用属性的名称。

NOTE

可以将任意数量的批注特性(自定义 XML 特性)应用于PropertyRef元素。然而,自定义特性可能不属于为 CSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示了一个PropertyRef元素,该元素用于通过引用EntityType元素上定义的属性来定义主键。

```
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

ReferentialConstraint 元素 (SSDL)

存储架构定义语言(SSDL)中的ReferentialConstraint元素表示基础数据库中的外键约束(也称为引用完整性约束)。约束的主体端和依赖端分别由 Principal 和 Dependent 子元素指定。通过 PropertyRef 元素引用参与主体端和依赖端的列。

ReferentialConstraint元素是 Association 元素的可选子元素。如果未使用ReferentialConstraint元素映射Association元素中指定的 foreign key 约束,则必须使用 AssociationSetMapping 元素来执行此操作。

ReferentialConstraint元素可以具有以下子元素:

- 文档(零个或一个)
- Principal(恰好一个)
- 依赖项(恰好一个)
- 批注元素(零个或多个)

适用的属性

可以将任意数量的批注特性(自定义 XML 特性)应用于ReferentialConstraint元素。然而,自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例演示一个Association元素，该元素使用ReferentialConstraint元素指定参与FK_CustomerOrders外键约束的列：

```
<Association Name="FK_CustomerOrders">
  <End Role="Customers"
    Type="ExampleModel.Store.Customers" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="Orders"
    Type="ExampleModel.Store.Orders" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Customers">
      <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
      <PropertyRef Name="CustomerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

ReturnType 元素 (SSDL)

存储架构定义语言(SSDL)中的ReturnType元素指定函数元素中定义的函数的返回类型。还可以使用ReturnType特性指定函数返回类型。

函数的返回类型是使用type特性或ReturnType元素指定的。

ReturnType元素可以具有以下子元素：

- CollectionType (—)

NOTE

可以向ReturnType元素应用任意数量的批注特性(自定义 XML 特性)。然而，自定义特性可能不属于为 SSDL 保留的任何 XML 命名空间。任何两个自定义特性的完全限定名称都不能相同。

示例

下面的示例使用一个函数，该函数返回行的集合。

```
<Function Name="GetProducts" IsComposable="true" Schema="dbo">
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="ProductID" Type="int" Nullable="false" />
        <Property Name="CategoryID" Type="bigint" Nullable="false" />
        <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
        <Property Name="UnitPrice" Type="money" />
        <Property Name="Discontinued" Type="bit" />
      </RowType>
    </CollectionType>
  </ReturnType>
</Function>
```

RowType 元素 (SSDL)

存储架构定义语言(SSDL)中的RowType元素将未命名的结构定义为在存储区中定义的函数的返回类型。

RowType元素是**CollectionType**元素的子元素：

RowType元素可以具有以下子元素：

- Property(一个或多个)

示例

下面的示例演示一个存储函数，该函数使用**CollectionType**元素来指定该函数返回行的集合(在**RowType**元素中指定)。

```
<Function Name="GetProducts" IsComposable="true" Schema="dbo">
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="ProductID" Type="int" Nullable="false" />
        <Property Name="CategoryID" Type="bigint" Nullable="false" />
        <Property Name="ProductName" Type="nvarchar" MaxLength="40" Nullable="false" />
        <Property Name="UnitPrice" Type="money" />
        <Property Name="Discontinued" Type="bit" />
      </RowType>
    </CollectionType>
  </ReturnType>
</Function>
```

Schema 元素 (SSDL)

存储架构定义语言(SSDL)中的**架构元素**是存储模型定义的根元素。它包括构成存储模型的对象、函数和容器的定义。

Schema元素可包含零个或多个下列表元素：

- 关联
- **EntityType**
- **EntityContainer**
- 函数

Schema元素使用**namespace**属性为存储模型中的实体类型和关联对象定义命名空间。在命名空间内，任何两个对象都不能同名。

存储模型命名空间不同于**Schema**元素的 XML 命名空间。存储模型命名空间(由命名空间特性定义)是实体类型和关联类型的逻辑容器。**Schema**元素的 XML 命名空间(由**xmlns**特性指示)是**schema**元素的子元素和属性的默认命名空间。格式为 <https://schemas.microsoft.com/ado/YYYY/MM/edm/ssdl> (其中，YYYY 和 MM 分别表示年份和月份)的 XML 命名空间是为 SSDL 保留的。自定义元素和特性不能位于具有此格式的命名空间中。

适用的属性

下表描述了可对**Schema**元素应用的属性。

名称	描述	值
Namespace	是	存储模型的命名空间。■属性的值用于构成类型的完全限定名称。例如，如果名为Customer的entitytype在位于examplemodel.store命名空间中，则entitytype的完全限定名称为位于examplemodel.store。 以下字符串不能用作Namespace特性的值：System、■或Edm。 Namespace属性的值不能与 CSDL Schema 元素中namespace属性的值相同。
Alias	是	用于取代命名空间名称的标识符。例如，如果名为Customer的EntityType位于位于 examplemodel.store 命名空间中，并且Alias属性的值为StorageModel.customer，则可以使用storageModel.customer 作为 EntityType 的完全限定名称 ■
■	是	数据提供程序。
ProviderManifestToken	是	一个标记，该标记指示提供程序清单返回到的提供程序。没有为该标记定义格式。标记的值由提供程序定义。有关 SQL Server 提供程序清单令牌的信息，请参阅 SqlClient for 实体框架。

示例

下面的示例演示一个架构元素，该元素包含一个EntityContainer元素、两个EntityType元素和一个Association元素。

```
<Schema Namespace="ExampleModel.Store"
  Alias="Self" Provider="System.Data.SqlClient"
  ProviderManifestToken="2008"
  xmlns="https://schemas.microsoft.com/ado/2009/11/edm/ssdl">
<EntityContainer Name="ExampleModelStoreContainer">
  <EntityType Name="Customers"
    EntityType="ExampleModel.Store.Customers"
    Schema="dbo" />
  <EntityType Name="Orders"
    EntityType="ExampleModel.Store.Orders"
    Schema="dbo" />
  <AssociationSet Name="FK_CustomerOrders"
    Association="ExampleModel.Store.FK_CustomerOrders">
    <End Role="Customers" EntitySet="Customers" />
    <End Role="Orders" EntitySet="Orders" />
  </AssociationSet>
</EntityContainer>
<EntityType Name="Customers">
  <Documentation>
    <Summary>Summary here.</Summary>
    <LongDescription>Long description here.</LongDescription>
  </Documentation>
  <Key>
    <PropertyRef Name="CustomerId" />
  </Key>
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <Property Name="Name" Type="nvarchar(max)" Nullable="false" />
</EntityType>
```

```

<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
    <Key>
        <PropertyRef Name="OrderId" />
    </Key>
    <Property Name="OrderId" Type="int" Nullable="false"
              c:CustomAttribute="someValue"/>
    <Property Name="ProductId" Type="int" Nullable="false" />
    <Property Name="Quantity" Type="int" Nullable="false" />
    <Property Name="CustomerId" Type="int" Nullable="false" />
    <c:CustomElement>
        Custom data here.
    </c:CustomElement>
</EntityType>
<Association Name="FK_CustomerOrders">
    <End Role="Customers"
        Type="ExampleModel.Store.Customers" Multiplicity="1">
        <OnDelete Action="Cascade" />
    </End>
    <End Role="Orders"
        Type="ExampleModel.Store.Orders" Multiplicity="*" />
<ReferentialConstraint>
    <Principal Role="Customers">
        <PropertyRef Name="CustomerId" />
    </Principal>
    <Dependent Role="Orders">
        <PropertyRef Name="CustomerId" />
    </Dependent>
</ReferentialConstraint>
</Association>
<Function Name="UpdateOrderQuantity"
          Aggregate="false"
          BuiltIn="false"
          NiladicFunction="false"
          IsComposable="false"
          ParameterTypeSemantics="AllowImplicitConversion"
          Schema="dbo">
    <Parameter Name="orderId" Type="int" Mode="In" />
    <Parameter Name="newQuantity" Type="int" Mode="In" />
</Function>
<Function Name="UpdateProductInOrder" IsComposable="false">
    <CommandText>
        UPDATE Orders
        SET ProductId = @productId
        WHERE OrderId = @orderId;
    </CommandText>
    <Parameter Name="productId"
              Mode="In"
              Type="int"/>
    <Parameter Name="orderId"
              Mode="In"
              Type="int"/>
</Function>
</Schema>

```

Annotation 特性

以存储架构定义语言 (SSDL) 表示的批注特性在存储模型中是自定义 XML 特性，这些特性提供有关存储模型中元素的额外元数据。除了具有有效的 XML 结构之外，以下约束也适用于批注特性：

- 批注特性不能位于为 SSDL 保留的任何 XML 命名空间中。
- 任何两个批注特性的完全限定名称都不能相同。

可以将多个批注特性应用于一个给定的 SSDL 元素。可以在运行时通过使用 System.web 命名空间中的类访问批注元素中包含的元数据。

示例

下面的示例演示一个 EntityType 元素，该元素具有应用于 "订单 id" 属性的批注特性。该示例还显示了添加到EntityType 元素的批注元素。

```
<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
  <Key>
    <PropertyRef Name="OrderId" />
  </Key>
  <Property Name="OrderId" Type="int" Nullable="false"
    c:CustomAttribute="someValue"/>
  <Property Name="ProductId" Type="int" Nullable="false" />
  <Property Name="Quantity" Type="int" Nullable="false" />
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <c:CustomElement>
    Custom data here.
  </c:CustomElement>
</EntityType>
```

批注元素 (SSDL)

以存储架构定义语言 (SSDL) 表示的批注元素是存储模型中的自定义 XML 元素，可提供有关存储模型的额外数据。除了具有有效的 XML 结构之外，批注元素还应满足以下约束：

- 批注元素不能位于为 SSDL 保留的任何 XML 命名空间中。
- 任何两个批注元素的完全限定名称都不能相同。
- 批注元素必须出现在给定 SSDL 元素的所有其他子元素之后。

多个批注元素可能是某个给定 SSDL 元素的子元素。从 .NET Framework 版本 4 开始，可以在运行时通过使用 System.web 命名空间中的类访问批注元素中包含的元数据。

示例

下面的示例显示具有 annotation 元素 (CustomElement) 的 EntityType 元素。该示例还显示了应用于 "订单 id" 属性的批注特性。

```
<EntityType Name="Orders" xmlns:c="http://CustomNamespace">
  <Key>
    <PropertyRef Name="OrderId" />
  </Key>
  <Property Name="OrderId" Type="int" Nullable="false"
    c:CustomAttribute="someValue"/>
  <Property Name="ProductId" Type="int" Nullable="false" />
  <Property Name="Quantity" Type="int" Nullable="false" />
  <Property Name="CustomerId" Type="int" Nullable="false" />
  <c:CustomElement>
    Custom data here.
  </c:CustomElement>
</EntityType>
```

方面 (SSDL)

以存储架构定义语言 (SSDL) 表示的方面表示对于 Property 元素中指定的列类型的约束。Facet 作为属性元素上的 XML 特性实现。

下表描述了 SSDL 中支持的方面：

■	指定在对属性值执行比较和排序操作时要使用的排序序列。
FixedLength	指定列值的长度是否可变。
MaxLength	指定列值的最大长度。
■	对于 Decimal 类型的属性, 指定属性值可以具有的位数。对于类型为 Time 、 DateTime 和 DateTimeOffset 的属性, 指定列值的秒小数部分的位数。
■	指定列值小数点右侧的位数。
Unicode	指示是否将列值存储为 Unicode。

定义查询 EF 设计器

2020/3/11 ·

本演练演示如何使用 EF 设计器将定义查询和相应的实体类型添加到模型。定义查询通常用于提供类似于数据库视图提供的功能，但该视图是在模型中定义的，而不是在数据库中定义的。使用定义查询可执行在 .edmx 文件的 **DefiningQuery** 元素中指定的 SQL 语句。有关详细信息，请参阅[SSDL 规范](#)中的 **DefiningQuery**。

使用定义查询时，还必须在模型中定义一个实体类型。实体类型用于呈现由定义查询公开的数据。请注意，通过此实体类型显示的数据是只读的。

无法将参数化查询作为定义查询执行。但是，可以通过将显示数据的实体类型的插入、更新和删除函数映射到存储过程来更新数据。有关详细信息，请参阅[对存储过程执行插入、更新和删除操作](#)。

本主题演示如何执行以下任务。

- 添加定义查询
- 向模型添加实体类型
- 将定义查询映射到实体类型

先决条件

若要完成此演练，您需要：

- Visual Studio 的最新版本。
- [School 示例数据库](#)。

设置项目

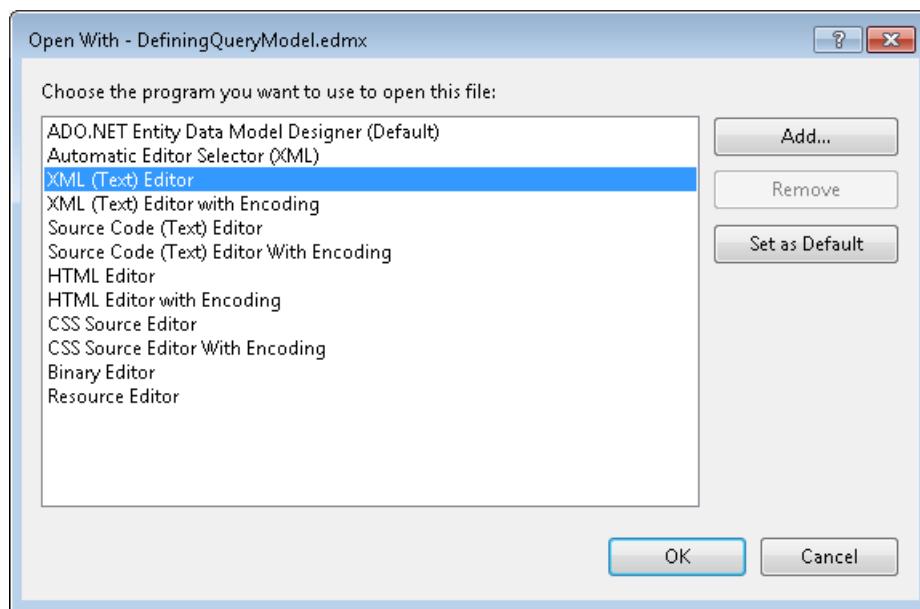
本演练使用 Visual Studio 2012 或更高版本。

- 打开 Visual Studio。
- 在“文件”菜单上，指向“新建”，再单击“项目”。
- 在左窗格中，单击“Visual C#”，然后选择“控制台应用程序”模板。
- 输入DefiningQuerySample作为项目名称，然后单击“确定”。

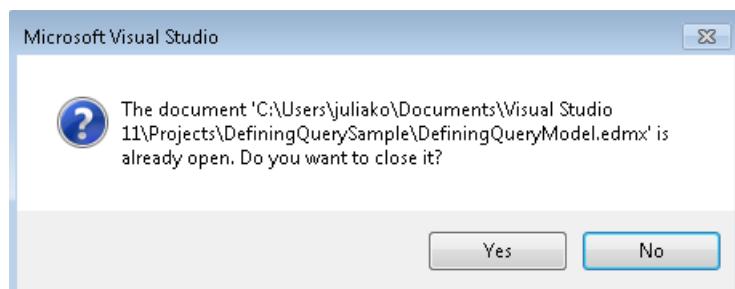
基于 School 数据库创建模型

- 右键单击“解决方案资源管理器”中的项目名称，指向“添加”，然后单击“新建项”。
- 从左侧菜单中选择“数据”，然后在“模板”窗格中选择“ADO.NET 实体数据模型”。
- 输入DefiningQueryModel作为文件名，然后单击“添加”。
- 在“选择模型内容”对话框中，选择“从数据库生成”，然后单击“下一步”。
- 单击“新建连接”。在“连接属性”对话框中，输入服务器名称（例如，(localdb)\mssqllocaldb），选择身份验证方法，为数据库名称键入 School，然后单击“确定”。“选择您的数据连接”对话框将通过数据库连接设置进行更新。
- 在“选择数据库对象”对话框中，检查“表”节点。这会将所有表添加到School模型。
- 单击“完成”。

- 在解决方案资源管理器中，右键单击**DefiningQueryModel**文件并选择“打开方式...”。
- 选择“XML（文本）编辑器”。



- 如果出现以下消息，请单击“是”：



添加定义查询

在此步骤中，我们将使用“XML 编辑器”向 .edmx 文件的 SSDL 部分添加定义查询和实体类型。

- 将 **EntitySet** 元素添加到 .edmx 文件的 SSDL 部分（第5行到第13行）。指定下列信息：
 - 仅指定 **EntitySet** 元素的 **名称** 和 **EntityType** 特性。
 - 实体类型的完全限定名称在 **EntityType** 特性中使用。
 - 要执行的 SQL 语句是在 **DefiningQuery** 元素中指定的。

```

<!-- SSDL content -->
<edmx:StorageModels>
    <Schema Namespace="SchoolModel.Store" Alias="Self" Provider="System.Data.SqlClient"
ProviderManifestToken="2008"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
xmlns="http://schemas.microsoft.com/ado/2009/11/edm/ssdl">
        <EntityContainer Name="SchoolModelStoreContainer">
            <EntityType Name="GradeReport" EntityType="SchoolModel.Store.GradeReport">
                <DefiningQuery>
                    SELECT CourseID, Grade, FirstName, LastName
                    FROM StudentGrade
                    JOIN
                    (SELECT * FROM Person WHERE EnrollmentDate IS NOT NULL) AS p
                    ON StudentID = p.PersonID
                </DefiningQuery>
            </EntityType>
            <EntityType Name="Course" EntityType="SchoolModel.Store.Course" store:Type="Tables" Schema="dbo" />
        </EntityContainer>
    </Schema>
</edmx:StorageModels>

```

- 将**EntityType**元素添加到 .EDMX 的 SSDL 部分。文件，如下所示。注意以下各项：
 - Name**特性的值对应于上面的**EntityType**元素中的**EntityType**特性的值，但该实体类型的完全限定名称在**EntityType**特性中使用。
 - 属性名称对应于**DefiningQuery**元素(如上所示)中的 SQL 语句返回的列名称。
 - 在此示例中，实体键由三个属性组成以确保唯一键值。

```

<EntityType Name="GradeReport">
    <Key>
        <PropertyRef Name="CourseID" />
        <PropertyRef Name="FirstName" />
        <PropertyRef Name="LastName" />
    </Key>
    <Property Name="CourseID"
        Type="int"
        Nullable="false" />
    <Property Name="Grade"
        Type="decimal"
        Precision="3"
        Scale="2" />
    <Property Name="FirstName"
        Type="nvarchar"
        Nullable="false"
        MaxLength="50" />
    <Property Name="LastName"
        Type="nvarchar"
        Nullable="false"
        MaxLength="50" />
</EntityType>

```

NOTE

如果以后运行■对话框，对存储模型所做的任何更改(包括定义查询)都将被覆盖。

向模型添加实体类型

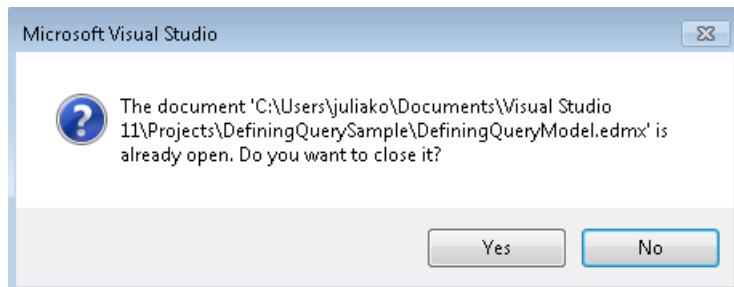
在此步骤中，我们将使用 EF 设计器将实体类型添加到概念模型。请注意以下事项：

- 实体的**名称**与上面的**EntityType**元素中的**EntityType**属性的值相对应。
- 属性名称对应于上面的**DefiningQuery**元素中的 SQL 语句返回的列名称。

- 在此示例中，实体键由三个属性组成以确保唯一键值。

在 EF 设计器中打开模型。

- 双击 "DefiningQueryModel"。
- 对以下消息说为 "是"：



此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。

- 右键单击设计器图面，然后选择 "添加新->"实体..."。
- 为 "键" 属性的 "实体名称" 和 "CourseID" 指定GradeReport。
- 右键单击 "GradeReport" 实体，然后选择 "添加新->"标量属性"。
- 将属性的默认名称更改为FirstName。
- 添加另一个标量属性，并为名称指定LastName。
- 添加另一个标量属性，并为该名称指定等级。
- 在 "属性" 窗口中，将评分的 "类型" 属性更改为Decimal。
- 选择FirstName和LastName属性。
- 在 "属性" 窗口中，将EntityKey属性值更改为True。

因此，将以下元素添加到 .edmx 文件的CSDL部分。

```
<EntitySet Name="GradeReport" EntityType="SchoolModel.GradeReport" />
<EntityType Name="GradeReport">
  ...
</EntityType>
```

将定义查询映射到实体类型

在此步骤中，我们将使用 "映射详细信息" 窗口来映射概念性实体类型和存储实体类型。

- 右键单击设计图面上的 "GradeReport" 实体，然后选择 "表映射"。
将显示 "映射详细信息" 窗口。
- 从 <添加表或视图> 下拉列表中选择 "GradeReport" (位于表s 下)。
显示概念和存储GradeReport实体类型之间的默认映射。

Parameter / Column	Operator	Property	Use Original...	Rows Affected Parameter
Functions				
Insert Using InsertPerson				
Parameters				
@ LastName : nvarchar	←	LastName : String	<input type="checkbox"/>	
@ FirstName : nvarchar	←	FirstName : String	<input type="checkbox"/>	
@ HireDate : datetime	←	HireDate : DateTime	<input type="checkbox"/>	
@ EnrollmentDate : datetime	←	EnrollmentDate : DateTime	<input type="checkbox"/>	
@ Discriminator : nvarchar	←	Discriminator : String	<input type="checkbox"/>	
Result Column Bindings				
NewPersonID	→	PersonID : Int32	<input checked="" type="checkbox"/>	

因此，`EntitySetMapping` 元素将添加到 .edmx 文件的映射部分。

```
<EntitySetMapping Name="GradeReports">
  <EntityTypeMapping TypeName="IsTypeOf(SchoolModel.GradeReport)">
    <MappingFragment StoreEntitySet="GradeReport">
      <ScalarProperty Name="LastName" ColumnName="LastName" />
      <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      <ScalarProperty Name="Grade" ColumnName="Grade" />
      <ScalarProperty Name="CourseID" ColumnName="CourseID" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

- 编译应用程序。

在代码中调用定义查询

现在可以使用`GradeReport`实体类型执行定义查询。

```
using (var context = new SchoolEntities())
{
  var report = context.GradeReports.FirstOrDefault();
  Console.WriteLine("{0} {1} got {2}",
    report.FirstName, report.LastName, report.Grade);
}
```

包含多个结果集的存储过程

2020/3/11 •

有时，在使用存储过程时，您需要返回多个结果集。此方案通常用于减少构成单个屏幕所需的数据库往返次数。在 EF5 之前，实体框架允许调用存储过程，但只将第一个结果集返回给调用代码。

本文将介绍两种方法，可用于从实体框架中的存储过程访问多个结果集。一个只使用代码，并处理代码 first 和 EF 设计器，另一个仅适用于 EF 设计器的。此方面的工具和 API 支持应在实体框架的未来版本中得以改善。

模型

本文中的示例使用的是基本的博客和文章模型，其中的博客包含多篇文章，张贴内容属于单个博客。我们将使用数据库中的存储过程，该存储过程返回所有博客和帖子，如下所示：

```
CREATE PROCEDURE [dbo].[GetAllBlogsAndPosts]
AS
    SELECT * FROM dbo.Blogs
    SELECT * FROM dbo.Posts
```

用代码访问多个结果集

我们可以执行使用代码发出原始 SQL 命令，以执行存储过程。此方法的优点是它适用于代码 first 和 EF 设计器。

为了获得多个结果集，我们需要使用 IObjectContextAdapter 接口删除 ObjectContext API。

获得 ObjectContext 后，我们可以使用转换方法将存储过程的结果转换为可在 EF 中跟踪和使用的实体。下面的代码示例演示如何执行此操作。

```

using (var db = new BloggingContext())
{
    // If using Code First we need to make sure the model is built before we open the connection
    // This isn't required for models created with the EF Designer
    db.Database.Initialize(force: false);

    // Create a SQL command to execute the sproc
    var cmd = db.Database.Connection.CreateCommand();
    cmd.CommandText = "[dbo].[GetAllBlogsAndPosts]";

    try
    {

        db.Database.Connection.Open();
        // Run the sproc
        var reader = cmd.ExecuteReader();

        // Read Blogs from the first result set
        var blogs = ((IObjectContextAdapter)db)
            .ObjectContext
            .Translate<Blog>(reader, "Blogs", MergeOption.AppendOnly);

        foreach (var item in blogs)
        {
            Console.WriteLine(item.Name);
        }

        // Move to second result set and read Posts
        reader.NextResult();
        var posts = ((IObjectContextAdapter)db)
            .ObjectContext
            .Translate<Post>(reader, "Posts", MergeOption.AppendOnly);

        foreach (var item in posts)
        {
            Console.WriteLine(item.Title);
        }
    }
    finally
    {
        db.Database.Connection.Close();
    }
}

```

转换方法接受我们在执行过程、EntitySet 名称和 MergeOption 时收到的读取器。EntitySet 名称将与派生上下文上的 DbSet 属性相同。MergeOption 枚举控制在内存中已存在相同的实体时如何处理结果。

在这里，我们将循环访问博客的集合，然后调用 NextResult，这对于上述代码非常重要，因为在移动到下一个结果集之前必须使用第一个结果集。

一旦调用这两个转换方法，EF 就会以与任何其他实体相同的方式跟踪博客和公告实体，因此可以修改或删除它们并将其保存为正常。

NOTE

当使用转换方法创建实体时，EF 不会考虑任何映射。它只是将结果集中的列名称与类的属性名进行匹配。

NOTE

这种情况下，如果启用了延迟加载，则访问其中一个博客实体上的“发布”属性后，EF 将连接到数据库以延迟加载所有发布，即使我们已经加载了所有发布。这是因为 EF 无法知道是否已加载所有发布或数据库中是否存在其他内容。如果要避免这种情况，则需要禁用延迟加载。

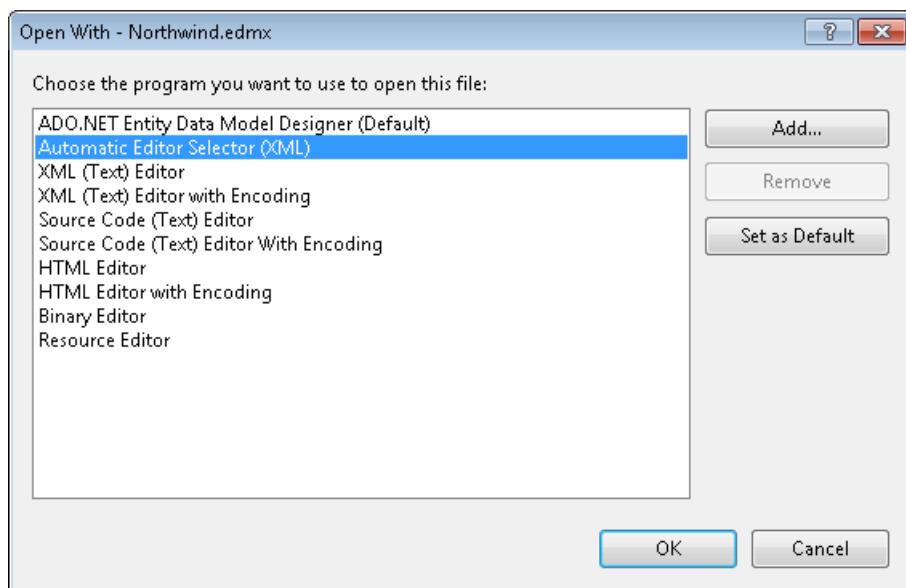
在 EDMX 中配置的多个结果集

NOTE

必须针对 .NET Framework 4.5，才能在 EDMX 中配置多个结果集。如果面向的是 .NET 4.0，则可以使用上一节中所示的基于代码的方法。

如果使用的是 EF 设计器，则还可以修改模型，使其了解将返回的不同结果集。在手头之前需要了解的一点是，该工具不是多个结果集感知，因此您需要手动编辑该 edmx 文件。编辑此类 edmx 文件的操作将起作用，但它也会破坏 VS 中模型的验证。如果验证模型，将始终会出现错误。

- 要执行此操作，您需要将该存储过程添加到您的模型中，就像对单个结果集查询执行此操作一样。
- 完成后，需要右键单击模型，然后选择“**打开方式**”。then **Xml**



将模型作为 XML 打开后，需要执行以下步骤：

- 在模型中查找复杂类型和函数导入：

```

<!-- CSDL content -->
<edmx:ConceptualModels>

...

<FunctionImport Name="GetAllBlogsAndPosts" ReturnType="Collection(BlogModel.GetAllBlogsAndPosts_Result)"
/>

...

<ComplexType Name=" GetAllBlogsAndPosts_Result">
<Property Type="Int32" Name="BlogId" Nullable="false" />
<Property Type="String" Name="Name" Nullable="false" MaxLength="255" />
<Property Type="String" Name="Description" Nullable="true" />
</ComplexType>

...

</edmx:ConceptualModels>

```

- 删除复杂类型
- 更新函数 import, 使其映射到你的实体, 在本例中, 它将如下所示:

```

<FunctionImport Name="GetAllBlogsAndPosts">
<ReturnType EntitySet="Blogs" Type="Collection(BlogModel.Blog)" />
<ReturnType EntitySet="Posts" Type="Collection(BlogModel.Post)" />
</FunctionImport>

```

这会告知模型, 存储过程将返回两个集合:一个博客项和一个发布项。

- 查找函数映射元素:

```

<!-- C-S mapping content -->
<edmx:Mappings>

...

<FunctionImportMapping FunctionImportName="GetAllBlogsAndPosts"
FunctionName="BlogModel.Store.GetAllBlogsAndPosts">
<ResultMapping>
<ComplexTypeMapping TypeName="BlogModel.GetAllBlogsAndPosts_Result">
<ScalarProperty Name="BlogId" ColumnName="BlogId" />
<ScalarProperty Name="Name" ColumnName="Name" />
<ScalarProperty Name="Description" ColumnName="Description" />
</ComplexTypeMapping>
</ResultMapping>
</FunctionImportMapping>

...

</edmx:Mappings>

```

- 将结果映射替换为每个要返回的实体, 如下所示:

```
<ResultMapping>
  <EntityTypeMapping TypeName = "BlogModel.Blog">
    <ScalarProperty Name="BlogId" ColumnName="BlogId" />
    <ScalarProperty Name="Name" ColumnName="Name" />
    <ScalarProperty Name="Description" ColumnName="Description" />
  </EntityTypeMapping>
</ResultMapping>
<ResultMapping>
  <EntityTypeMapping TypeName="BlogModel.Post">
    <ScalarProperty Name="BlogId" ColumnName="BlogId" />
    <ScalarProperty Name="PostId" ColumnName="PostId"/>
    <ScalarProperty Name="Title" ColumnName="Title" />
    <ScalarProperty Name="Text" ColumnName="Text" />
  </EntityTypeMapping>
</ResultMapping>
```

还可以将结果集映射到复杂类型，如默认创建的类型。为此，您创建了一个新的复杂类型，而不是删除它们，并使用在上述示例中使用了实体名称的任何位置的复杂类型。

更改这些映射后，你可以保存模型并执行以下代码以使用该存储过程：

```
using (var db = new BlogEntities())
{
  var results = db.GetAllBlogsAndPosts();

  foreach (var result in results)
  {
    Console.WriteLine("Blog: " + result.Name);
  }

  var posts = results.GetNextResult<Post>();

  foreach (var result in posts)
  {
    Console.WriteLine("Post: " + result.Title);
  }

  Console.ReadLine();
}
```

NOTE

如果手动编辑模型的 edmx 文件，则从数据库重新生成模型时，它将被覆盖。

摘要

这里，我们介绍了两种不同的方法来使用实体框架访问多个结果集。这两种方法都是相同的，具体取决于你的情况和首选项，你应选择最适合你的环境的方法。它计划在未来版本的实体框架中对多个结果集的支持进行改进，并且不再需要执行本文档中的步骤。

表值函数 (TVF)

2020/3/11 ·

NOTE

EF5 ■-实体框架5中引入了本页中所述的功能、api 等。如果使用的是早期版本，则部分或全部信息不适用。

视频和分步演练演示了如何使用 Entity Framework Designer 映射表值函数(TVF)。它还演示了如何从 LINQ 查询调用 TVF。

TVF 当前仅在 Database First 工作流中受支持。

实体框架版本5中引入了 TVF 支持。请注意，若要使用表值函数、枚举和空间类型等新功能，则必须以 .NET Framework 4.5 为目标。默认情况下，Visual Studio 2012 面向 .NET 4.5。

TVF 非常类似于具有一个关键区别的存储过程：TVF 的结果是可组合的。这意味着，可以在 LINQ 查询中使用 TVF 的结果，而存储过程的结果不能。

观看视频

主讲人 :Julia Kornich

[WMV](#) | [MP4](#) | [WMV \(ZIP\)](#)

先决条件

若要完成本演练，你需要：

- 安装[School 数据库](#)。
- 具有最新版本的 Visual Studio

设置项目

1. 打开 Visual Studio
2. 在 "文件" 菜单上，指向 "新建"，然后单击 "项目"
3. 在左窗格中，单击 "Visual C#"，然后选择控制台模板
4. 输入TVF作为项目名称，然后单击 "确定"

将 TVF 添加到数据库

- 选择 " > SQL Server 对象资源管理器
- 如果 LocalDB 不在服务器列表中：右键单击SQL Server，然后选择 "添加 SQL Server 使用默认Windows 身份验证连接到 LocalDB 服务器"
- 展开 LocalDB 节点
- 在 "数据库" 节点下，右键单击 "School" 数据库节点，然后选择 "新建查询 ... "
- 在 T-sql 编辑器中粘贴以下 TVF 定义

```
CREATE FUNCTION [dbo].[GetStudentGradesForCourse]
(@CourseID INT)
RETURNS TABLE
RETURN
SELECT [EnrollmentID],
[CourseID],
[StudentID],
[Grade]
FROM [dbo].[StudentGrade]
WHERE CourseID = @CourseID
```

- 在 T-sql 编辑器上单击鼠标右键按钮，然后选择 "执行"
- GetStudentGradesForCourse 函数将添加到 School 数据库

创建模型

- 右键单击 "解决方案资源管理器" 中的项目名称，指向 "添加"，然后单击 "新建项"
- 从左侧菜单中选择 "数据"，然后在 "模板" 窗格中选择 "ADO.NET 实体数据模型"
- 输入 TVFModel 作为文件名，然后单击 "添加"
- 在 "选择模型内容" 对话框中，选择 "从数据库生成"，然后单击 "下一步"
- 单击 "新建连接" 输入 (localdb)\Mssqllocaldb 在 "服务器名称" 文本框中输入 School 数据库名称，单击 "确定"
- 在 "选择数据库对象" 对话框中，在 "表" 节点下，选择 "Person"、"StudentGrade" 和 "课程" 表
- 从 Visual Studio 2012 开始，选择 "存储过程" 和 "函数" 节点下的 GetStudentGradesForCourse 函数，该函数从 Visual Studio 开始，Entity Designer 允许批处理导入存储过程和函数
- 单击 "完成"
- 此时会显示 Entity Designer，它提供了用于编辑模型的设计图面。您在 "选择数据库对象" 对话框中选择的所有对象都将添加到模型中。
- 默认情况下，每个导入的存储过程或函数的结果形状将自动成为实体模型中的新复杂类型。但我们想要将 GetStudentGradesForCourse 函数的结果映射到 StudentGrade 实体：右键单击设计图面，然后在 "模型浏览器" 中选择 "模型浏览器"，选择 "函数导入"，然后在 "编辑函数导入" 对话框中双击 "GetStudentGradesForCourse" 函数，选择 "实体" 并选择 StudentGrade

保留和检索数据

打开定义 Main 方法的文件。将以下代码添加到 Main 函数中。

下面的代码演示如何生成使用表值函数的查询。该查询将结果投影到一个匿名类型中，该类型包含相关课程标题和一个等级大于或等于 3.5 的相关学生。

```
using (var context = new SchoolEntities())
{
    var CourseID = 4022;
    var Grade = 3.5M;

    // Return all the best students in the Microeconomics class.
    var students = from s in context.GetStudentGradesForCourse(CourseID)
                  where s.Grade >= Grade
                  select new
                  {
                      s.Person,
                      s.Course.Title
                  };

    foreach (var result in students)
    {
        Console.WriteLine(
            "Couse: {0}, Student: {1} {2}",
            result.Title,
            result.Person.FirstName,
            result.Person.LastName);
    }
}
```

编译并运行该应用程序。该程序生成以下输出：

```
Couse: Microeconomics, Student: Arturo Anand
Couse: Microeconomics, Student: Carson Bryant
```

Summary

在本演练中，我们介绍了如何使用 Entity Framework Designer 映射表值函数(Tvf)。它还演示了如何从 LINQ 查询调用 TVF。

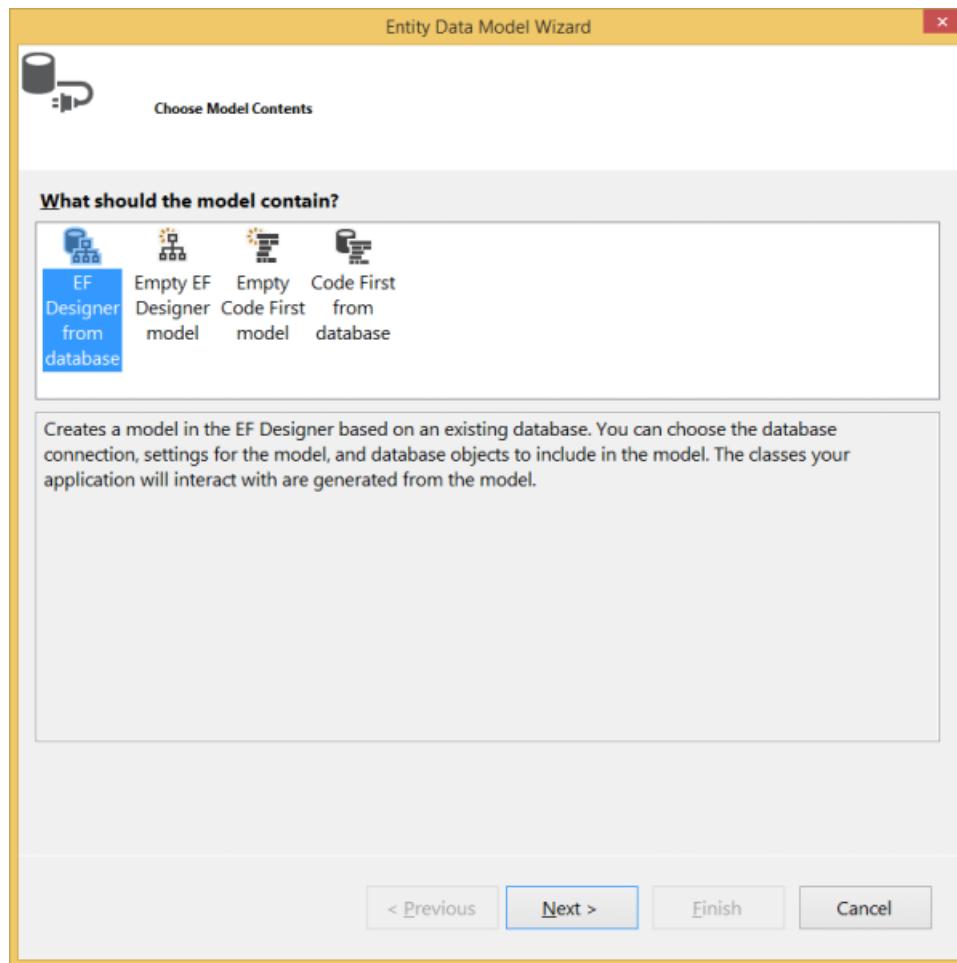
Entity Framework Designer 键盘快捷方式

2020/3/11 •

本页提供适用于 Visual Studio Entity Framework Tools 的各种屏幕中的键盘 shorcuts 列表。

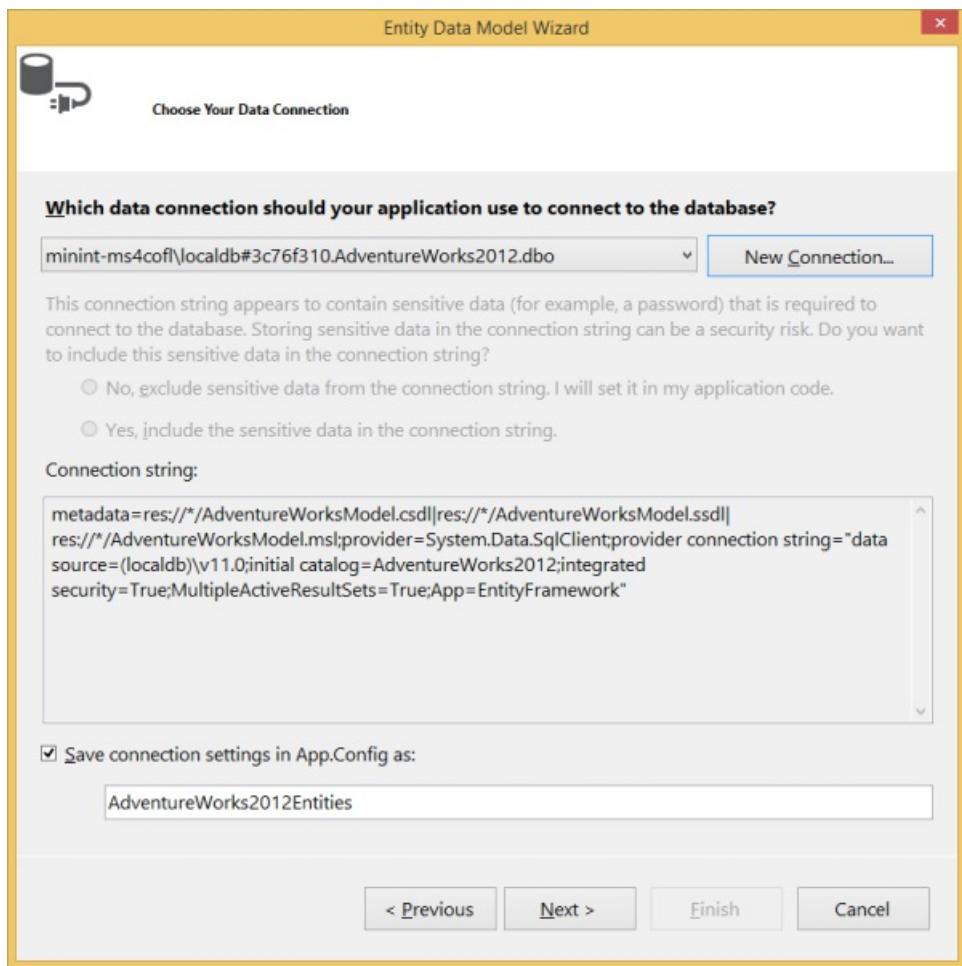
ADO.NET 实体数据模型向导

第一步：选择模型内容



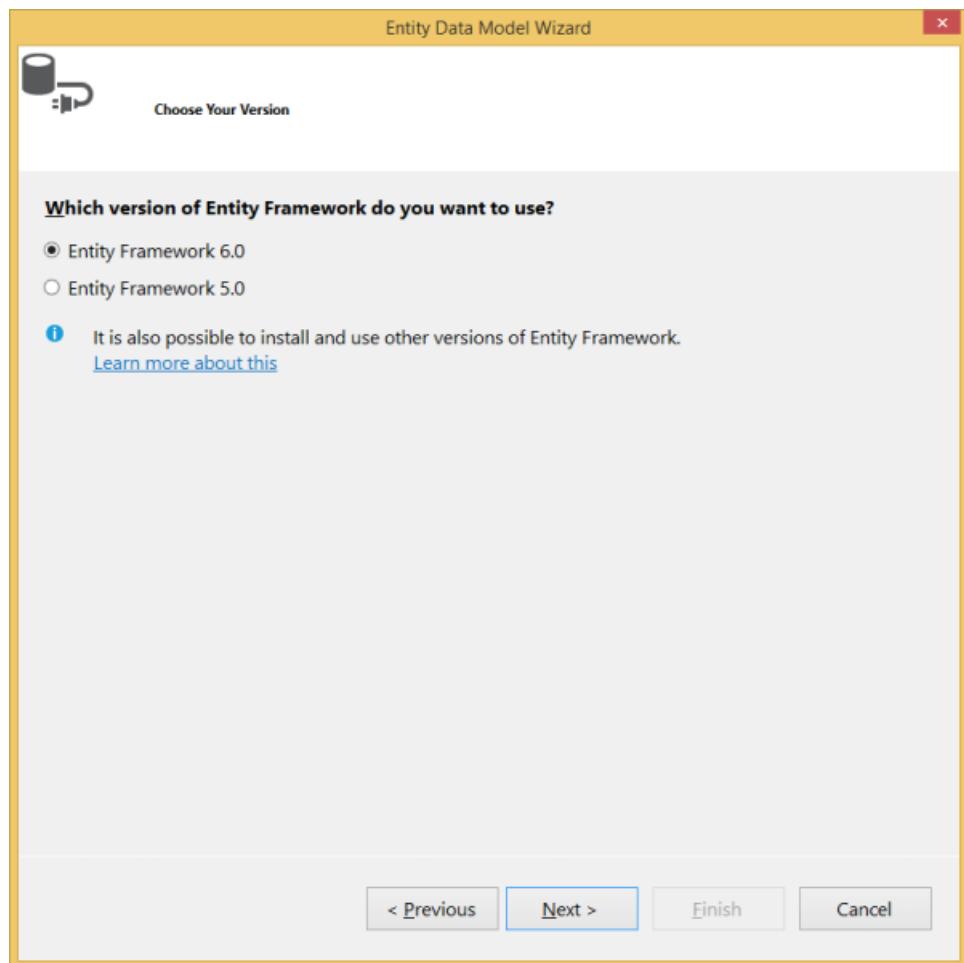
功能	操作	说明
Alt + n	转到下一个屏幕	不适用于所有模型内容选择。
Alt + f	结束向导	不适用于所有模型内容选择。
Alt + w	将焦点切换到“模型应包含哪些内容？”窗格中显示与该命名空间关联的连接字符串和其他元数据。	

步骤2：选择连接



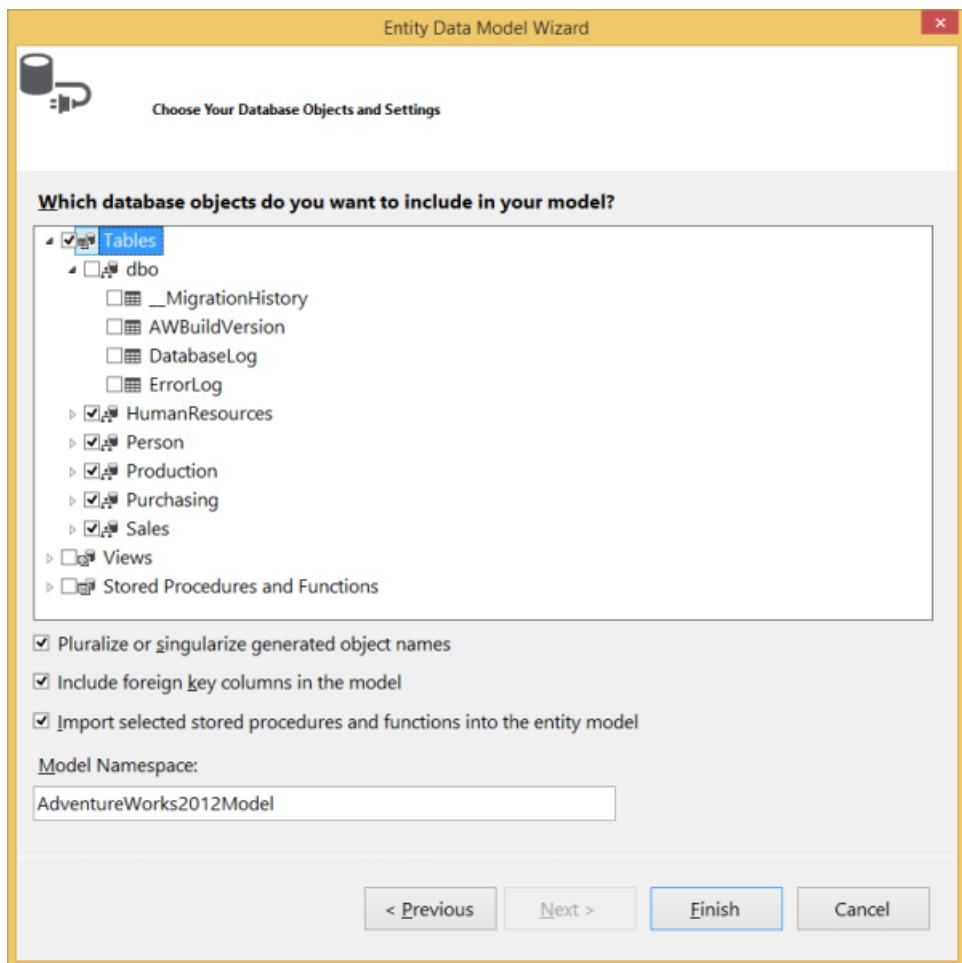
Alt + n	转到下一个屏幕	
Alt + p	移到上一个屏幕	
Alt + w	将焦点切换到 "模型应包含哪些内容？" 窗格中显示与该命名空间关联的连接字符串和其他元数据。	
Alt + c	打开 "连接属性" 窗口	允许定义新数据库连接。
Alt + e	从连接字符串中排除敏感数据	
Alt + i	在连接字符串中包含敏感数据	
Alt + s	切换 "保存 App.config 中的连接设置" 选项	

步骤3: 选择你的版本



操作	功能	说明
Alt + n	转到下一个屏幕	
Alt + p	移到上一个屏幕	
Alt + w	将焦点切换到实体框架版本选择	允许指定不同版本的实体框架以便在项目中使用。

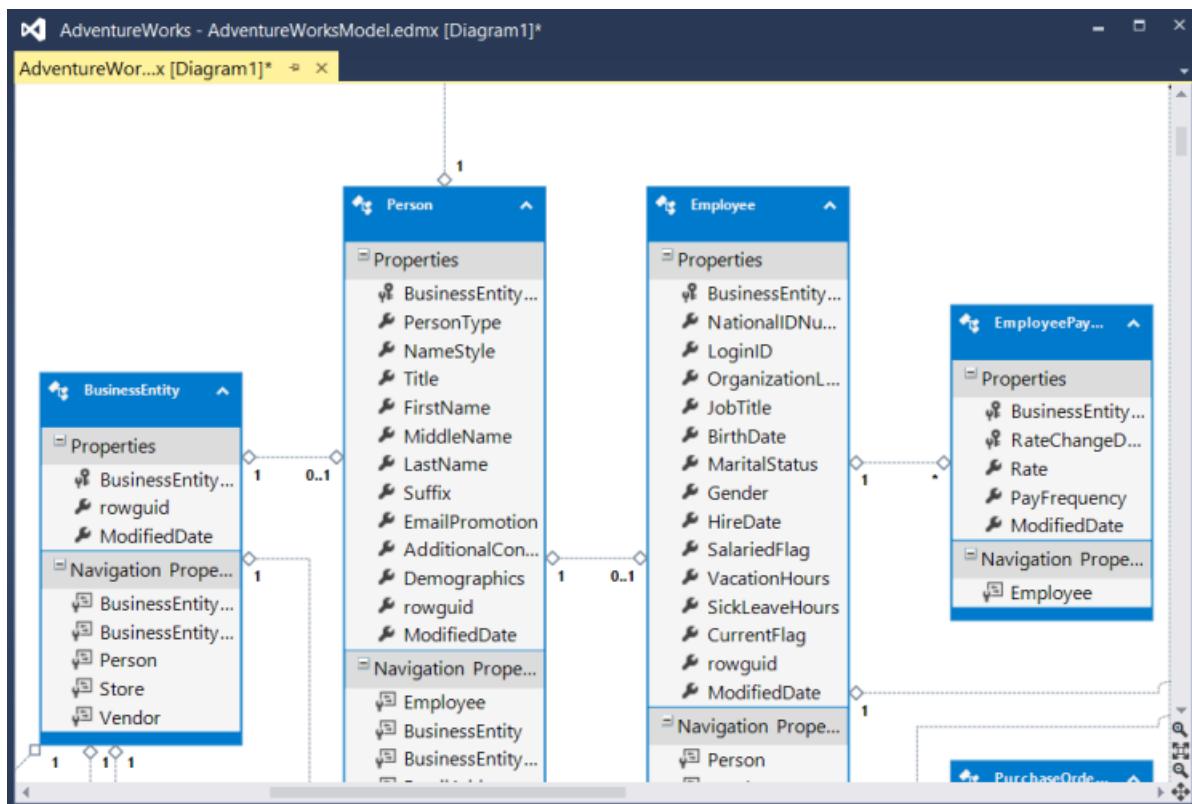
步骤4: 选择数据库对象和设置



Alt + f	结束向导	
Alt + p	移到上一个屏幕	
Alt + w	将焦点切换到数据库对象选择窗格	允许指定要进行反向工程的数据库对象。
Alt + s	切换 "复数形式或确定所生成的对象名称" 选项	
Alt + k	切换 "在模型中包括外键列" 选项	不适用于所有模型内容选择。
Alt + i	切换 "将选定的存储过程和函数导入实体模型" 选项	不适用于所有模型内容选择。
Alt + m	将焦点切换到 "模型命名空间" 文本字段	不适用于所有模型内容选择。
空格键	切换元素上的选定内容	如果元素有子级，则将同时切换所有子元素
Left	折叠子树	
Right	展开子树	
Up	在树中导航到上一个元素	

■	导航到树中的下一个元素	
---	-------------	--

EF 设计器图面



■/Enter	切换选定内容	通过焦点切换对象上的选定内容。
Esc	取消选择	取消当前选择。
Ctrl + A	全选	选择设计图面上的所有形状。
■	上移	将所选实体向上移动一个网格增量。 如果在列表中，则移动到上一个同级子字段。
■	下移	将所选实体向下移动一个网格增量。 如果在列表中，则移动到下一个同级子字段。
■	左移	将所选实体向左移动一个网格增量。 如果在列表中，则移动到上一个同级子字段。
■	右移	将所选实体向右移动一个网格增量。 如果在列表中，则移动到下一个同级子字段。
Shift + ■	左侧大小形状	将所选实体的宽度降低一个网格增量。

命令	功能	说明
Shift + ■	向右调整形状大小	将所选实体的宽度增加一个网格增量。
■	第一个对等	将焦点和所选内容移到设计图面上处于同一对等级别的第一个对象。
End	上一个对等	将焦点和所选内容移到设计图面上处于同一对等级别的最后一个对象。
Ctrl + Home	第一对等(焦点)	与第一个对等方相同,但是移动焦点,而不是移动焦点和选择。
Ctrl + End	上一个对等(焦点)	与上一对等方相同,但是移动焦点,而不是移动焦点和选择。
Tab	下一个对等	将焦点和所选内容移到设计图面上处于同一对等级别的下一个对象。
Shift+Tab	上一个对等	将焦点和选择内容移到设计图面上同一对等级别上的上一个对象。
Alt + Ctrl + Tab	下一个对等(焦点)	与下一个对等方相同,但是移动焦点,而不是移动焦点和选择。
Alt + Ctrl + Shift + Tab	上一个对等(焦点)	与上一对等方相同,但是移动焦点,而不是移动焦点和选择。
<	递增	移动到层次结构中较高级别的设计图面上的下一个对象。如果层次结构中此形状上没有任何形状(即,对象直接放置在设计图面上),则选择该关系图。
>	降	在设计图面上,将其移动到层次结构中这一级别下的下一个包含对象。如果没有包含对象,则这是一个无操作。
Ctrl + <	递增(焦点)	与“递增”命令相同,但移动时无需选择。
Ctrl + >	降序(焦点)	与“向下”命令相同,但移动时不选择。
Shift + End	遵循连接	在实体中,移动到此实体所连接到的实体。
Delete	删除	从关系图中删除对象或连接器。
■	插入	当选择“标量属性”隔离舱头或属性本身时,将新属性添加到实体。
Pg	向上滚动关系图	将设计图面向上滚动,增加等于当前可见设计图面的高度的75%。
■	向下滚动关系图	向下滚动设计图面。
Shift + ■	向右滚动图示	将设计图面向右滚动。

Shift + ■	向左滚动图示	向左滚动设计图面。
F2	进入编辑模式	用于进入文本控件的编辑模式的标准键盘快捷键。
Shift + F10	显示快捷菜单	用于显示选定项的快捷菜单的标准键盘快捷方式。
Ctrl + Shift + ■ Ctrl + Shift + ■	语义放大	在鼠标指针下方的关系图视图区域中放大。
Ctrl + Shift + ■ ■ ctrl + Shift + ■	语义缩小	缩小鼠标指针下关系图视图的区域。当您缩小当前关系图中心时，它会将关系图重新居中。
Control + Shift + "+" ■ + ■	放大	放大关系图视图的中心。
Control + Shift + "-" ■ + ■	缩小	从关系图视图的单击区域缩小。当您缩小当前关系图中心时，它会将关系图重新居中。
■ ctrl ■■■	缩放区域	放大所选区域的中心。按住 Ctrl + Shift 键的同时，您会看到光标变为放大镜，这允许您定义要缩小到的区域。
■■■ + ' i '	打开映射详细信息窗口	打开“映射详细信息”窗口以编辑所选实体的映射

“映射详细信息”窗口

The screenshot shows the 'Mapping Details - Person' window. The left sidebar has a tree view with 'Tables' expanded, showing 'Maps to Person' and 'Column Mappings'. Under 'Column Mappings', there is a list of columns from the 'Person' table being mapped to properties in the 'BusinessEntity' table. Each mapping consists of a source column name and its target property name.

Column	Operat...	Value / Property
BusinessEntityID : int	↳	BusinessEntityID : Int32
PersonType : nchar	↳	PersonType : String
NameStyle : bit	↳	NameStyle : Boolean
Title : nvarchar	↳	Title : String
FirstName : nvarchar	↳	FirstName : String
MiddleName : nvarchar	↳	MiddleName : String
LastName : nvarchar	↳	LastName : String
Suffix : nvarchar	↳	Suffix : String
EmailPromotion : int	↳	EmailPromotion : Int32
AdditionalContactInfo : xml	↳	AdditionalContactInfo : String
Demographics : xml	↳	Demographics : String
rowguid : uniqueidentifier	↳	rowguid : Guid
ModifiedDate : datetime	↳	ModifiedDate : DateTime

Tab	切换上下文	在主窗口区域和左侧工具栏之间切换
-----	-------	------------------

■	导航	在主窗口区域中上下移动行，或在列上向右和向左移动。在左侧工具栏中的按钮之间移动。
Enter 空格键	选择	选择左侧工具栏中的按钮。
Alt + ■	打开列表	如果选定的单元格包含下拉列表，则下拉列表。
Enter	列表选择	选择下拉列表中的元素。
Esc	列表关闭	关闭下拉列表。

Visual Studio 导航

实体框架还提供了多个可以映射自定义键盘快捷方式的操作（默认情况下不映射快捷方式）。若要创建这些自定义快捷方式，请单击“工具”菜单，然后单击“选项”。在“环境”下，选择“键盘”。在中间向下滚动列表，直到可以选择所需的命令，在“按快捷键”文本框中输入该快捷方式，然后单击“分配”。可能的快捷方式如下：

OtherContextMenus. MicrosoftDataEntityDesignContext. ComplexProperty. ComplexTypes
OtherContextMenus.MicrosoftDataEntityDesignContext.AddCodeGenerationItem
OtherContextMenus.MicrosoftDataEntityDesignContext.AddFunctionImport
OtherContextMenus. MicrosoftDataEntityDesignContext. AddEnumType
OtherContextMenus. MicrosoftDataEntityDesignContext
OtherContextMenus. MicrosoftDataEntityDesignContext. ComplexProperty
OtherContextMenus. MicrosoftDataEntityDesignContext
OtherContextMenus. MicrosoftDataEntityDesignContext. ScalarProperty
OtherContextMenus.MicrosoftDataEntityDesignContext.AddNewDiagram
OtherContextMenus.MicrosoftDataEntityDesignContext.AddtoDiagram
OtherContextMenus. MicrosoftDataEntityDesignContext. ■

||||

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus.MicrosoftDataEntityDesignContext.ConverttoEnum

OtherContextMenus. MicrosoftDataEntityDesignContext. CollapseAll

OtherContextMenus. MicrosoftDataEntityDesignContext. ■

OtherContextMenus. MicrosoftDataEntityDesignContext. ExportasImage

OtherContextMenus. MicrosoftDataEntityDesignContext. LayoutDiagram

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext. FunctionImportMapping

OtherContextMenus.MicrosoftDataEntityDesignContext.GenerateDatabasefromModel

OtherContextMenus.MicrosoftDataEntityDesignContext.GoToDefinition

OtherContextMenus. MicrosoftDataEntityDesignContext. ShowGrid

OtherContextMenus MicrosoftDataEntityDesignContext

OtherContextMenus.MicrosoftDataEntityDesignContext.IncludeRelated

OtherContextMenus.MicrosoftDataEntityDesignContext.MappingDetails

OtherContextMenus. MicrosoftDataEntityDesignContext. Modelbrowser ■ modelementbrowser

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveDiagramstoSeparateFile

OtherContextMenus. MicrosoftDataEntityDesignContext. MoveProperties

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Down5

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.ToBottom

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.ToTop

OtherContextMenus. MicrosoftDataEntityDesignContext. MoveProperties

OtherContextMenus.MicrosoftDataEntityDesignContext.MoveProperties.Up5

OtherContextMenus.MicrosoftDataEntityDesignContext.MovetonewDiagram

||||

OtherContextMenus. MicrosoftDataEntityDesignContext■

OtherContextMenus. MicrosoftDataEntityDesignContext. MovetoNewComplexType

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus.MicrosoftDataEntityDesignContext.RemovefromDiagram

OtherContextMenus. MicrosoftDataEntityDesignContext. ■

OtherContextMenus. MicrosoftDataEntityDesignContext. ScalarPropertyFormat

OtherContextMenus.MicrosoftDataEntityDesignContext.ScalarPropertyFormat.DisplayNameandType

OtherContextMenus. MicrosoftDataEntityDesignContext. ■ BaseType

OtherContextMenus. MicrosoftDataEntityDesignContext. ■

OtherContextMenus. MicrosoftDataEntityDesignContext. ■

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext. SelectAll

OtherContextMenus.MicrosoftDataEntityDesignContext.SelectAssociation

OtherContextMenus.MicrosoftDataEntityDesignContext.ShowinDiagram

OtherContextMenus.MicrosoftDataEntityDesignContext.ShowinModelBrowser

OtherContextMenus. MicrosoftDataEntityDesignContext.StoredProcedureMapping

OtherContextMenus. MicrosoftDataEntityDesignContext. TableMapping

OtherContextMenus.MicrosoftDataEntityDesignContext.UpdateModelfromDatabase

OtherContextMenus. MicrosoftDataEntityDesignContext. Validate

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

||||

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext. 50

OtherContextMenus MicrosoftDataEntityDesignContext

OtherContextMenus MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext

OtherContextMenus. MicrosoftDataEntityDesignContext. ZoomIn

OtherContextMenus. MicrosoftDataEntityDesignContext. ZoomOut

OtherContextMenus. MicrosoftDataEntityDesignContext. ZoomToFit

■ EntityDataModelBrowser

■ EntityDataModelMappingDetails

查询和查找实体

2020/4/8 ·

本主题介绍使用实体框架查询数据的各种方法，包括 LINQ 和 Find 方法。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

使用查询查找实体

DbSet 和 IDbSet 可实现 IQueryable，因此可用作针对数据库编写 LINQ 查询的起点。在此不适合深入讨论 LINQ，但以下提供了几个简单示例：

```
using (var context = new BloggingContext())
{
    // Query for all blogs with names starting with B
    var blogs = from b in context.Blogs
                where b.Name.StartsWith("B")
                select b;

    // Query for the Blog named ADO.NET Blog
    var blog = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .FirstOrDefault();
}
```

请注意， DbSet 和 IDbSet 始终针对数据库创建查询，并且始终会涉及数据库往返，即使返回的实体已存在于上下文中。出现以下情况时，会针对数据库执行查询：

- 查询由 foreach (C#) 或 For Each (Visual Basic) 语句枚举。
- 查询由集合操作（如 [ToArray](#)、[ToDictionary](#) 或 [ToList](#)）枚举。
- 在查询最外部指定了 LINQ 运算符，例如 [First](#) 或 [Any](#)。
- 调用了以下方法：DbSet 上的 [Load](#) 扩展方法、[DbEntityEntry.Reload](#) 和 [Database.ExecuteSqlCommand](#)。

从数据库返回结果时，上下文中不存在的对象会附加到上下文。如果某个对象已存在于上下文中，则不返回现有对象（不会使用数据库值覆盖该对象的属性在对应项中的当前值和原始值）。

执行查询时，结果集中不回返回已添加到上下文但尚未保存到数据库的实体。若要获取上下文中的数据，请参阅[本地数据](#)。

如果查询未从数据库返回任何行，则结果将是空集合，而不是 NULL。

使用主键查找实体

DbSet 上的 Find 方法使用主键值来尝试查找由上下文跟踪的实体。如果在上下文中未找到实体，则会向数据库发送查询以在其中查找实体。如果未在上下文中或数据库中找到实体，则返回 NULL。

Find 与使用查询有两个重要区别：

- 只有未在上下文中找到具有给定键的实体时，才会往返数据库。
- Find 会返回处于“已添加”状态的实体。即 Find 会返回已添加到上下文但尚未保存到数据库的实体。

通过主键查找实体

以下代码演示了 Find 的部分用法：

```
using (var context = new BloggingContext())
{
    // Will hit the database
    var blog = context.Blogs.Find(3);

    // Will return the same instance without hitting the database
    var blogAgain = context.Blogs.Find(3);

    context.Blogs.Add(new Blog { Id = -1 });

    // Will find the new blog even though it does not exist in the database
    var newBlog = context.Blogs.Find(-1);

    // Will find a User which has a string primary key
    var user = context.Users.Find("johndoe1987");
}
```

通过组合主键查找实体

实体框架允许实体具有组合键，即由多个属性组成的键。例如，用户可具有表示特定博客的用户设置的 BlogSettings 实体。因为用户的每个博客只能拥有一个 BlogSettings，所以可以选择将 BlogId 和用户名结合作为 BlogSettings 的主键。以下代码尝试查找 BlogId = 3，用户名 = "johndoe1987" 的 BlogSettings：

```
using (var context = new BloggingContext())
{
    var settings = context.BlogSettings.Find(3, "johndoe1987");
}
```

请注意，具有组合键时，需使用 ColumnAttribute 或 Fluent API 来指定组合键属性的顺序。指定构成键的值时，对 Find 的调用必须使用此顺序。

加载方法

2020/3/11 •

在某些情况下，你可能需要将实体从数据库加载到上下文中，而不会立即对这些实体执行任何操作。这是一个很好的例子，就是加载用于数据绑定的实体，如[本地数据](#)中所述。实现此目的的一个常见方法是编写 LINQ 查询，然后对其调用 System.Linq.Enumerable.ToList，只是立即放弃创建的列表。负载扩展方法的工作方式与 System.Linq.Enumerable.ToList 相同，只是它可以避免完全创建列表。

本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

下面是两个使用 Load 的示例。第一种方式是从 Windows 窗体数据绑定应用程序中获取，在将其绑定到本地集合之前，将使用 Load 查询实体，如[本地数据](#)中所述：

```
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);

    _context = new ProductContext();

    _context.Categories.Load();
    categoryBindingSource.DataSource = _context.Categories.Local.ToBindingList();
}
```

第二个示例演示如何使用 Load 加载相关实体的筛选集合，如[加载相关实体](#)中所述：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog
    context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();
}
```

本地数据

2020/3/11 •

直接对 DbSet 运行 LINQ 查询将始终向数据库发送查询，但你可以使用 DbSet 属性访问当前在内存中的数据。使用 DbContext 和 DbContext 方法，还可以访问额外的信息 EF 正在跟踪实体。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

使用本地查看本地数据

DbSet 的 Local 属性提供对当前正在由上下文跟踪并且未标记为已删除的集的实体的简单访问。访问本地属性绝不会使查询发送到数据库。这意味着它通常在查询已执行后使用。负载扩展方法可用于执行查询，以便上下文可以跟踪结果。例如：

```
using (var context = new BloggingContext())
{
    // Load all blogs from the database into the context
    context.Blogs.Load();

    // Add a new blog to the context
    context.Blogs.Add(new Blog { Name = "My New Blog" });

    // Mark one of the existing blogs as Deleted
    context.Blogs.Remove(context.Blogs.Find(1));

    // Loop over the blogs in the context.
    Console.WriteLine("In Local:");
    foreach (var blog in context.Blogs.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            blog.BlogId,
            blog.Name,
            context.Entry(blog).State);
    }

    // Perform a query against the database.
    Console.WriteLine("\nIn DbSet query:");
    foreach (var blog in context.Blogs)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            blog.BlogId,
            blog.Name,
            context.Entry(blog).State);
    }
}
```

如果数据库中有两个博客—"ADO.NET Blog" 的 BlogId 为1, "Visual Studio 博客" 的 BlogId 为2, 则可能会收到以下输出：

```
In Local:  
Found 0: My New Blog with state Added  
Found 2: The Visual Studio Blog with state Unchanged
```

```
In DbSet query:  
Found 1: ADO.NET Blog with state Deleted  
Found 2: The Visual Studio Blog with state Unchanged
```

这说明了三个要点：

- 新博客的 "我的新博客" 包含在本地集合中，即使尚未将其保存到数据库中。此博客的主键为零，因为数据库尚未生成实体的实际密钥。
- 即使上下文仍在跟踪 "ADO.NET Blog"，也不会将其包含在本地集合中。这是因为我们从 DbSet 中删除了它，从而将其标记为已删除。
- 当使用 DbSet 来执行查询时，将在结果中包含标记为删除的博客(ADO.NET 博客)，并且未保存到数据库中的新博客(我的新博客)不会包含在结果中。这是因为，DbSet 正在对数据库执行查询，并且返回的结果始终反映了数据库中的内容。

使用本地来添加和删除上下文中的实体

DbSet 上的本地属性返回一个 [ObservableCollection](#)，其中包含与该上下文的内容保持同步的事件。这意味着可以在本地集合或 DbSet 中添加或移除实体。这也意味着，将新实体引入上下文的查询将导致用这些实体更新本地集合。例如：

```

using (var context = new BloggingContext())
{
    // Load some posts from the database into the context
    context.Posts.Where(p => p.Tags.Contains("entity-framework")).Load();

    // Get the local collection and make some changes to it
    var localPosts = context.Posts.Local;
    localPosts.Add(new Post { Name = "What's New in EF" });
    localPosts.Remove(context.Posts.Find(1));

    // Loop over the posts in the context.
    Console.WriteLine("In Local after entity-framework query: ");
    foreach (var post in context.Posts.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            post.Id,
            post.Title,
            context.Entry(post).State);
    }

    var post1 = context.Posts.Find(1);
    Console.WriteLine(
        "State of post 1: {0} is {1}",
        post1.Name,
        context.Entry(post1).State);

    // Query some more posts from the database
    context.Posts.Where(p => p.Tags.Contains("asp.net")).Load();

    // Loop over the posts in the context again.
    Console.WriteLine("\nIn Local after asp.net query: ");
    foreach (var post in context.Posts.Local)
    {
        Console.WriteLine(
            "Found {0}: {1} with state {2}",
            post.Id,
            post.Title,
            context.Entry(post).State);
    }
}

```

假设我们有几个标记有 "entity-框架" 和 "asp.net" 的文章，输出可能如下所示：

```

In Local after entity-framework query:
Found 3: EF Designer Basics with state Unchanged
Found 5: EF Code First Basics with state Unchanged
Found 0: What's New in EF with state Added
State of post 1: EF Beginners Guide is Deleted

In Local after asp.net query:
Found 3: EF Designer Basics with state Unchanged
Found 5: EF Code First Basics with state Unchanged
Found 0: What's New in EF with state Added
Found 4: ASP.NET Beginners Guide with state Unchanged

```

这说明了三个要点：

- 添加到本地集合中的新 post "EF 新增功能" 将由上下文在已添加状态中进行跟踪。因此，当调用 SaveChanges 时，它将被插入到数据库中。
- 从本地集合中删除的 post (EF 初学者指南)现在已在上下文中标记为 "已删除"。因此，当调用 SaveChanges 时，将从数据库中删除该方法。
- 将第二个查询加载到上下文中的附加 post (ASP.NET 初学者指南)自动添加到本地集合中。

关于本地需要注意的最后一点是，因为这是一个 ObservableCollection 的性能，不适用于大量实体。因此，如果你在上下文中处理数千个实体，则可能不建议使用本地。

使用本地进行 WPF 数据绑定

DbSet 上的本地属性可以直接用于 WPF 应用程序中的数据绑定，因为它是 ObservableCollection 的实例。如前面部分所述，这意味着它将与上下文内容自动保持同步，并且上下文的内容将自动与它保持同步。请注意，您需要预先填充本地集合，其中包含的数据可以绑定到任何内容，因为本地从不会导致数据库查询。

这不是完整的 WPF 数据绑定示例的合适位置，但关键元素是：

- 设置绑定源
- 将其绑定到集的本地属性
- 使用数据库的查询填充本地。

WPF 绑定到导航属性

如果要执行主/详细数据绑定，则可能需要将详细信息视图绑定到某个实体的导航属性。若要执行此操作，一种简单的方法是将 ObservableCollection 用于导航属性。例如：

```
public class Blog
{
    private readonly ObservableCollection<Post> _posts =
        new ObservableCollection<Post>();

    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual ObservableCollection<Post> Posts
    {
        get { return _posts; }
    }
}
```

使用 Local 在 SaveChanges 中清理实体

在大多数情况下，从导航属性中删除的实体在上下文中不会自动标记为 "已删除"。例如，如果从博客发布了一个 Post 对象，则在调用 SaveChanges 后，将不会自动删除此 post。如果需要删除此实体，则可能需要在调用 SaveChanges 之前，或将其标记为已删除，然后再将其标记为已删除。例如：

```
public override int SaveChanges()
{
    foreach (var post in this.Posts.Local.ToList())
    {
        if (post.Blog == null)
        {
            this.Posts.Remove(post);
        }
    }

    return base.SaveChanges();
}
```

上面的代码使用本地集合查找所有帖子，并将没有博客引用的任何文章标记为已删除。

System.Linq.Enumerable.ToList 调用是必需的，因为在枚举时，删除调用会对集合进行修改。在大多数其他情况下，您可以直接对本地属性进行查询，而不先使用 System.Linq.Enumerable.ToList。

为 Windows 窗体数据绑定使用本地和对 `IEnumerable<DbSet>` 获得

Windows 窗体不支持直接使用 `ObservableCollection` 的完全保真数据绑定。不过，你仍然可以使用 `DbSet` 本地属性进行数据绑定，以获得前面几节中所述的全部权益。这是通过对 `IEnumerable<DbSet>` 获得扩展方法实现的，该方法创建由本地 `ObservableCollection` 支持的 `IBindingList` 实现。

此位置不适合完全 Windows 窗体的数据绑定示例，但关键元素是：

- 设置对象绑定源
- 使用对 `IEnumerable<DbSet>` 获得()将其绑定到集的本地属性
- 使用数据库的查询填充本地

获取有关被跟踪实体的详细信息

此系列中的许多示例使用 `Entry` 方法返回实体的 `DbEntityEntry` 实例。然后，此 `entry` 对象充当用于收集有关实体的信息（如当前状态）的起始点，以及用于对实体执行操作（如显式加载相关实体）的起始点。

条目方法为上下文跟踪的多个或所有实体返回 `DbEntityEntry` 对象。这允许您收集信息或对多个实体执行操作，而不是只收集单个条目。例如：

```
using (var context = new BloggingContext())
{
    // Load some entities into the context
    context.Blogs.Load();
    context.Authors.Load();
    context.Readers.Load();

    // Make some changes
    context.Blogs.Find(1).Title = "The New ADO.NET Blog";
    context.Blogs.Remove(context.Blogs.Find(2));
    context.Authors.Add(new Author { Name = "Jane Doe" });
    context.Readers.Find(1).Username = "johndoe1987";

    // Look at the state of all entities in the context
    Console.WriteLine("All tracked entities: ");
    foreach (var entry in context.ChangeTracker.Entries())
    {
        Console.WriteLine(
            "Found entity of type {0} with state {1}",
            ObjectContext.GetObjectType(entry.Entity.GetType()).Name,
            entry.State);
    }

    // Find modified entities of any type
    Console.WriteLine("\nAll modified entities: ");
    foreach (var entry in context.ChangeTracker.Entries()
        .Where(e => e.State == EntityState.Modified))
    {
        Console.WriteLine(
            "Found entity of type {0} with state {1}",
            ObjectContext.GetObjectType(entry.Entity.GetType()).Name,
            entry.State);
    }

    // Get some information about just the tracked blogs
    Console.WriteLine("\nTracked blogs: ");
    foreach (var entry in context.ChangeTracker.Entries<Blog>())
    {
        Console.WriteLine(
            "Found Blog {0}: {1} with original Name {2}",
            entry.Entity.BlogId,
            entry.Entity.Name,
            entry.Property(p => p.Name).OriginalValue);
    }

    // Find all people (author or reader)
    Console.WriteLine("\nPeople: ");
    foreach (var entry in context.ChangeTracker.Entries<IPerson>())
    {
        Console.WriteLine("Found Person {0}", entry.Entity.Name);
    }
}
```

你会注意到，我们在示例中引入了一个作者和读者类，这两个类都实现 IPerson 接口。

```

public class Author : IPerson
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public string Biography { get; set; }
}

public class Reader : IPerson
{
    public int ReaderId { get; set; }
    public string Name { get; set; }
    public string Username { get; set; }
}

public interface IPerson
{
    string Name { get; }
}

```

假设数据库中包含以下数据：

BlogId = 1 和 Name = 'ADO.NET Blog' 的博客
 BlogId = 2 的博客和名称 = 'Visual Studio 博客'
 BlogId = 3 且名称为 ".NET Framework 博客" 的博客
 作者为 AuthorId = 1, 名称 = "Joe Bloggs"
 ReaderId = 1 且名称 = "John Doe" 的读取器

运行代码的输出为：

```

All tracked entities:
Found entity of type Blog with state Modified
Found entity of type Blog with state Deleted
Found entity of type Blog with state Unchanged
Found entity of type Author with state Unchanged
Found entity of type Author with state Added
Found entity of type Reader with state Modified

All modified entities:
Found entity of type Blog with state Modified
Found entity of type Reader with state Modified

Tracked blogs:
Found Blog 1: The New ADO.NET Blog with original Name ADO.NET Blog
Found Blog 2: The Visual Studio Blog with original Name The Visual Studio Blog
Found Blog 3: .NET Framework Blog with original Name .NET Framework Blog

People:
Found Person John Doe
Found Person Joe Bloggs
Found Person Jane Doe

```

这些示例阐释了几个要点：

- 条目方法返回所有状态中实体的条目，包括 "已删除"。将此与本地（不包括已删除的实体）进行比较。
- 当使用非泛型条目方法时，将返回所有实体类型的条目。当使用泛型条目方法时，仅为作为泛型类型的实例的实体返回条目。上述内容用于获取所有博客的条目。它还用于获取实现 IPerson 的所有实体的条目。这表明泛型类型不一定是实际的实体类型。
- LINQ to Objects 可用于筛选返回的结果。在上述情况下，只要修改了任何类型的实体，就可以找到它。

请注意，`DbEntityEntry` 实例始终包含非 null 的实体。关系项和存根项不表示为 `DbEntityEntry` 实例，因此无需对其进行筛选。

非跟踪查询

2020/3/11 •

有时，可能需要从查询中获取实体，但不能通过上下文跟踪这些实体。在只读方案中查询大量实体时，这可能会导致更好的性能。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

新的扩展方法 `AsNoTracking` 允许以这种方式运行任何查询。例如：

```
using (var context = new BloggingContext())
{
    // Query for all blogs without tracking them
    var blogs1 = context.Blogs.AsNoTracking();

    // Query for some blogs without tracking them
    var blogs2 = context.Blogs
        .Where(b => b.Name.Contains(".NET"))
        .AsNoTracking()
        .ToList();
}
```

原生 SQL 查询

2020/3/11 ·

实体框架允许使用 LINQ 和实体类进行查询。但是，有时您可能想要对数据库直接使用原始 SQL 来运行查询。这包括调用存储过程，这对于当前不支持映射到存储过程的 Code First 模型非常有用。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

为实体编写 SQL 查询

DbSet 上的 `SqlQuery` 方法允许编写原始 SQL 查询来返回实体实例。返回的对象将由上下文跟踪，就像它们是通过 LINQ 查询返回的一样。例如：

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();
}
```

请注意，就像 LINQ 查询一样，查询在枚举结果之前不会执行，在上面的示例中，此操作是通过调用 `System.Linq.Enumerable.ToList` 来完成的。

出于两个原因而编写原始 SQL 查询时应谨慎。首先，应编写查询以确保它仅返回真正请求类型的实体。例如，使用继承等功能时，可以轻松编写一个查询，该查询将创建错误的 CLR 类型的实体。

其次，某些类型的原始 SQL 查询会暴露潜在的安全风险，尤其是围绕 SQL 注入式攻击。确保在查询中使用参数，以防止此类攻击。

从存储过程加载实体

可以使用 DbSet 从存储过程的结果中加载实体。例如，下面的代码调用 `dbo` 数据库中的 `GetBlogs` 过程：

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs.SqlQuery("dbo.GetBlogs").ToList();
}
```

还可以使用以下语法将参数传递给存储过程：

```
using (var context = new BloggingContext())
{
    var blogId = 1;

    var blogs = context.Blogs.SqlQuery("dbo.GetBlogById @p0", blogId).Single();
}
```

为非实体类型编写 SQL 查询

返回任何类型的实例的 SQL 查询（包括基元类型）都可以在数据库类上使用 `SqlQuery` 方法创建。例如：

```
using (var context = new BloggingContext())
{
    var blogNames = context.Database.SqlQuery<string>(
        "SELECT Name FROM dbo.Blogs").ToList();
}
```

即使对象是实体类型的实例，上下文也永远不会跟踪从数据库的 `SqlQuery` 返回的结果。

向数据库发送原始命令

可以使用数据库上的 `ExecuteSqlCommand` 方法将非查询命令发送到数据库。例如：

```
using (var context = new BloggingContext())
{
    context.Database.ExecuteSqlCommand(
        "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
}
```

请注意，使用 `ExecuteSqlCommand` 对数据库中的数据所做的任何更改在从数据库中加载或重新加载实体之前，对上下文的更改是不透明的。

输出参数

如果使用了 `output` 参数，则在完全读取结果之前，它们的值将不可用。这是因为 `DbDataReader` 的基础行为，有关详细信息，请参阅[使用 DataReader 检索数据](#)。

正在加载相关实体

2020/3/11 •

实体框架支持三种方法来加载相关数据-预先加载、延迟加载和显式加载。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

积极加载

预先加载是指一种实体类型的查询，它还会在查询中加载相关实体。预先加载是通过使用 `Include` 方法实现的。例如，下面的查询将加载博客以及与每个博客相关的所有帖子。

```
using (var context = new BloggingContext())
{
    // Load all blogs and related posts.
    var blogs1 = context.Blogs
        .Include(b => b.Posts)
        .ToList();

    // Load one blog and its related posts.
    var blog1 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include(b => b.Posts)
        .FirstOrDefault();

    // Load all blogs and related posts
    // using a string to specify the relationship.
    var blogs2 = context.Blogs
        .Include("Posts")
        .ToList();

    // Load one blog and its related posts
    // using a string to specify the relationship.
    var blog2 = context.Blogs
        .Where(b => b.Name == "ADO.NET Blog")
        .Include("Posts")
        .FirstOrDefault();
}
```

NOTE

`Include` 是 `System.web` 命名空间中的扩展方法，因此请确保正在使用该命名空间。

积极加载多个级别

还可以积极加载多个级别的相关实体。下面的查询显示如何为收集和引用导航属性执行此操作的示例。

```
using (var context = new BloggingContext())
{
    // Load all blogs, all related posts, and all related comments.
    var blogs1 = context.Blogs
        .Include(b => b.Posts.Select(p => p.Comments))
        .ToList();

    // Load all users, their related profiles, and related avatar.
    var users1 = context.Users
        .Include(u => u.Profile.Avatar)
        .ToList();

    // Load all blogs, all related posts, and all related comments
    // using a string to specify the relationships.
    var blogs2 = context.Blogs
        .Include("Posts.Comments")
        .ToList();

    // Load all users, their related profiles, and related avatar
    // using a string to specify the relationships.
    var users2 = context.Users
        .Include("Profile.Avatar")
        .ToList();
}
```

NOTE

目前不能筛选要加载的相关实体。Include 将始终引入所有相关实体。

延迟加载

延迟加载是指首次访问引用实体/实体的属性时，从数据库自动加载实体或实体集合的过程。使用 POCO 实体类型时，延迟加载是通过创建派生代理类型的实例，然后重写虚拟属性来添加加载挂钩来实现的。例如，在使用下面定义的博客实体类时，将在第一次访问“发布”导航属性时加载相关的文章：

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public virtual ICollection<Post> Posts { get; set; }
}
```

为序列化关闭延迟加载

延迟加载和序列化不会很好地混合，如果您不小心，只是因为启用了延迟加载，最终就可以对整个数据库进行查询。大多数序列化程序通过访问类型实例上的每个属性来工作。属性访问会触发延迟加载，因此会序列化更多的实体。在这些实体上，将访问这些实体的属性，甚至还会加载更多实体。在对实体进行序列化之前，最好关闭延迟加载。以下部分介绍了如何执行该操作。

关闭特定导航属性的延迟加载

可以通过将“Post”属性设为“非虚拟”来关闭 Post 集合的延迟加载：

```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }
    public string Tags { get; set; }

    public ICollection<Post> Posts { get; set; }
}
```

仍可使用预先加载(请参阅上面的[积极加载](#))或 Load 方法(请参阅下面的[显式加载](#))来加载发布集合。

关闭所有实体的延迟加载

通过在配置属性上设置标志, 可以为上下文中的所有实体关闭延迟加载。例如:

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

仍可使用预先加载(请参阅上面的[积极加载](#))或 Load 方法(请参阅下面的[显式加载](#))来加载相关实体。

显式加载

即使已禁用延迟加载, 仍可能会延迟加载相关实体, 但必须通过显式调用来完成此操作。为此, 请对相关实体的条目使用 Load 方法。例如:

```
using (var context = new BloggingContext())
{
    var post = context.Posts.Find(2);

    // Load the blog related to a given post.
    context.Entry(post).Reference(p => p.Blog).Load();

    // Load the blog related to a given post using a string.
    context.Entry(post).Reference("Blog").Load();

    var blog = context.Blogs.Find(1);

    // Load the posts related to a given blog.
    context.Entry(blog).Collection(p => p.Posts).Load();

    // Load the posts related to a given blog
    // using a string to specify the relationship.
    context.Entry(blog).Collection("Posts").Load();
}
```

NOTE

如果实体具有指向另一个实体的导航属性, 则应使用 Reference 方法。另一方面, 如果实体具有指向其他实体的集合的导航属性, 则应使用集合方法。

显式加载相关实体时应用筛选器

查询方法提供对实体框架在加载相关实体时将使用的基础查询的访问权限。然后, 你可以使用 LINQ 将筛选器应用于查询, 然后通过调用 LINQ 扩展方法(例如 System.Linq.Enumerable.ToList、Load 等)执行这些筛选器。查询方法

既可以与引用导航属性一起使用，也可以用于集合导航属性，但对于可用于仅加载部分集合的集合最有用。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Load the posts with the 'entity-framework' tag related to a given blog.
    context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();

    // Load the posts with the 'entity-framework' tag related to a given blog
    // using a string to specify the relationship.
    context.Entry(blog)
        .Collection("Posts")
        .Query()
        .Where(p => p.Tags.Contains("entity-framework"))
        .Load();
}
```

使用查询方法时，通常最好关闭导航属性的延迟加载。这是因为在执行筛选的查询之前或之后，延迟加载机制可能会自动加载整个集合。

NOTE

虽然可将关系指定为字符串而不是 lambda 表达式，但在使用字符串时返回的 IQueryable 并不是泛型的，因此通常需要强制转换方法，然后才能对其执行任何有用的操作。

使用 Query 计算相关实体而不加载它们

有时，了解数据库中与另一个实体相关的实体的数量并不实际产生加载所有这些实体的成本是非常有用的。带有 LINQ Count 方法的 Query 方法可用于执行此操作。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Count how many posts the blog has.
    var postCount = context.Entry(blog)
        .Collection(b => b.Posts)
        .Query()
        .Count();
}
```

使用实体框架 6 保存数据

2020/4/8 •

在本节中，可以找到有关 EF 的更改跟踪功能的信息，以及在调用 `SaveChanges` 将对象的任何更改保存到数据库时会发生的情况。

自动检测更改

2020/3/11 ·

使用最多 POCO 实体时，会通过检测更改算法来处理实体如何更改（以及需要向数据库发送更新）。检测更改的工作方式是检测实体的当前属性值与在查询或附加实体时存储在快照中的原始属性值之间的差异。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

默认情况下，在调用以下方法时，实体框架自动执行检测更改：

- DbSet.Find
- DbSet
- DbSet
- DbSet.AddRange
- DbSet
- DbSet.RemoveRange
- DbSet
- DbContext.SaveChanges
- DbContext.GetValidationErrors
- DbContext.Entry
- DbChangeTracker

禁用更改的自动检测

如果正在跟踪上下文中的大量实体，并在循环中多次调用其中一种方法，则可以通过在循环的持续时间内关闭更改检测来显著提高性能。例如：

```
using (var context = new BloggingContext())
{
    try
    {
        context.Configuration.AutoDetectChangesEnabled = false;

        // Make many calls in a loop
        foreach (var blog in aLotOfBlogs)
        {
            context.Blogs.Add(blog);
        }
    }
    finally
    {
        context.Configuration.AutoDetectChangesEnabled = true;
    }
}
```

不要忘记重新启用循环后的更改检测，我们使用了 try/finally 来确保始终重新启用该循环，即使循环中的代码引发异常也是如此。

禁用和重新启用的替代方法是始终始终关闭更改的自动检测，并调用上下文。ChangeTracker 显式或使用更改跟踪代理。这两个选项都是高级选项，可轻松地在应用程序中引入微妙 bug，因此请谨慎使用。

如果需要在上下文中添加或删除多个对象，请考虑使用 AddRange 和 DbSet RemoveRange。此方法在添加或删除操作完成后，只自动检测更改一次。

使用实体状态

2020/3/11 ·

本主题将介绍如何添加实体并将其附加到上下文，以及实体框架如何在 SaveChanges 期间处理这些实体。实体框架负责在实体连接到上下文时跟踪实体的状态，但在断开连接或 N 层方案中，你可以让 EF 知道实体应采用的状态。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

实体状态和 SaveChanges

实体可以是 EntityState 枚举定义的五种状态之一。这些状态是：

- 已添加：上下文正在跟踪实体，但数据库中尚不存在该实体
- 保持不变：上下文正在跟踪实体，该实体存在于数据库中，并且其属性值未更改为数据库中的值
- 已修改：实体正在由上下文跟踪，并存在于数据库中，并且其部分或全部属性值已修改
- 已删除：实体正在由上下文跟踪，并存在于数据库中，但在下次调用 SaveChanges 时已标记为要从数据库中删除
- 已分离：上下文未跟踪该实体

对于不同状态中的实体，SaveChanges 执行不同的操作：

- SaveChanges 不会接触到未更改的实体。对于处于未更改状态的实体，不会将更新发送到数据库。
- 添加的实体将插入到数据库中，并在 SaveChanges 返回时变为不变。
- 修改后的实体将在数据库中更新，并在 SaveChanges 返回时变得不变。
- 删除的实体将从数据库中删除，然后与上下文分离。

下面的示例演示了如何更改实体或实体关系图的状态。

向上下文中添加新实体

可以通过对 DbSet 调用 Add 方法，将新实体添加到上下文中。这会使实体处于已添加状态，这意味着它将在下一次调用 SaveChanges 时插入到数据库中。例如：

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Blogs.Add(blog);
    context.SaveChanges();
}
```

向上下文中添加新实体的另一种方法是将其状态更改为“已添加”。例如：

```
using (var context = new BloggingContext())
{
    var blog = new Blog { Name = "ADO.NET Blog" };
    context.Entry(blog).State = EntityState.Added;
    context.SaveChanges();
}
```

最后，可以通过将新实体挂钩到已跟踪的另一个实体来向上下文中添加新实体。这可以是将新实体添加到另一个实体的集合导航属性中，或通过设置另一个实体的引用导航属性来指向新实体。例如：

```
using (var context = new BloggingContext())
{
    // Add a new User by setting a reference from a tracked Blog
    var blog = context.Blogs.Find(1);
    blog.Owner = new User { UserName = "johndoe1987" };

    // Add a new Post by adding to the collection of a tracked Blog
    blog.Posts.Add(new Post { Name = "How to Add Entities" });

    context.SaveChanges();
}
```

请注意，对于所有这些示例，如果添加的实体具有对尚未跟踪的其他实体的引用，则这些新实体也将添加到上下文中，并将在下次调用 SaveChanges 时插入到数据库中。

将现有实体附加到上下文

如果你的实体已存在于数据库中，但当前未由上下文跟踪，则可以使用 DbSet 上的 Attach 方法告诉上下文跟踪实体。实体在上下文中将处于未更改状态。例如：

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);

    // Do some more work...

    context.SaveChanges();
}
```

请注意，如果在调用 SaveChanges 时未执行附加实体的任何其他操作，则不会对数据库进行任何更改。这是因为实体处于未更改状态。

将现有实体附加到上下文的另一种方法是将其状态更改为“未更改”。例如：

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

请注意，对于这两个示例，如果附加的实体已引用尚未跟踪的其他实体，则这些新实体还会附加到处于未更改状态的上下文。

将现有的但修改的实体附加到上下文

如果你的实体已存在于数据库中，但可能已对其进行了更改，则可以通知上下文附加实体并将其状态设置为“已修改”。例如：

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Entry(existingBlog).State = EntityState.Modified;

    // Do some more work...

    context.SaveChanges();
}
```

当你将状态更改为 "已修改" 时，实体的所有属性都将标记为已修改，并且在调用 `SaveChanges` 时，所有属性值都将发送到数据库。

请注意，如果附加的实体具有对尚未跟踪的其他实体的引用，则这些新实体将以未更改状态附加到上下文中，而不会自动进行修改。如果需要将多个实体标记为 "已修改"，则应单独设置每个实体的状态。

更改所跟踪实体的状态

您可以通过对其项设置状态属性来更改已被跟踪的实体的状态。例如：

```
var existingBlog = new Blog { BlogId = 1, Name = "ADO.NET Blog" };

using (var context = new BloggingContext())
{
    context.Blogs.Attach(existingBlog);
    context.Entry(existingBlog).State = EntityState.Unchanged;

    // Do some more work...

    context.SaveChanges();
}
```

请注意，为已跟踪的实体调用 "添加或附加" 也可用于更改实体状态。例如，为当前处于添加状态的实体调用 `Attach` 会将其状态更改为 "不更改"。

插入或更新模式

某些应用程序的一种常见模式是将实体添加为新实体(生成数据库插入)，或将实体附加到现有实体并将其标记为已修改(生成数据库更新)，具体取决于主键的值。例如，在使用数据库生成的整数主键时，通常将包含零键的实体作为新的和具有非零键的实体视为现有的。可以通过基于主键值的检查设置实体状态来实现此模式。例如：

```
public void InsertOrUpdate(Blog blog)
{
    using (var context = new BloggingContext())
    {
        context.Entry(blog).State = blog.BlogId == 0 ?
            EntityState.Added :
            EntityState.Modified;

        context.SaveChanges();
    }
}
```

请注意，当你将状态更改为 "已修改" 时，实体的所有属性都将标记为已修改，并且在调用 `SaveChanges` 时，所有属性值都将发送到数据库。

使用属性值

2020/3/11 ·

大多数情况下实体框架将负责跟踪实体实例的属性的状态、原始值和当前值。但是，在某些情况下（例如，已断开连接的情况下），你希望查看或操作有关属性的信息 EF。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

实体框架跟踪所跟踪实体的每个属性的两个值。当前值为，如名称所示，是实体中属性的当前值。原始值是从数据库中查询实体或将实体附加到上下文时属性所具有的值。

使用属性值的一般机制有两种：

- 单个属性的值可以使用属性方法以强类型方式获取。
- 实体的所有属性的值都可以读取到 `DbPropertyValues` 对象中。然后，`DbPropertyValues` 充当类似字典的对象，以允许读取和设置属性值。`DbPropertyValues` 对象中的值可以从其他 `DbPropertyValues` 对象中的值或其他某个对象的值进行设置，如实体的另一个副本或简单的数据传输对象 (DTO)。

以下部分显示了使用上述两种机制的示例。

获取和设置单个属性的当前值或原始值

下面的示例演示如何读取属性的当前值，然后将其设置为新值：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(3);

    // Read the current value of the Name property
    string currentName1 = context.Entry(blog).Property(u => u.Name).CurrentValue;

    // Set the Name property to a new value
    context.Entry(blog).Property(u => u.Name).CurrentValue = "My Fancy Blog";

    // Read the current value of the Name property using a string for the property name
    object currentName2 = context.Entry(blog).Property("Name").CurrentValue;

    // Set the Name property to a new value using a string for the property name
    context.Entry(blog).Property("Name").CurrentValue = "My Boring Blog";
}
```

使用 `OriginalValue` 属性而非 `CurrentValue` 属性来读取或设置原始值。

请注意，当使用字符串指定属性名时，返回的值将被类型化为 "object"。另一方面，如果使用 lambda 表达式，则返回值为强类型。

如果新值不同于旧值，则将属性值设置为时，只会将属性标记为已修改。

以这种方式设置属性值时，即使关闭了 `AutoDetectChanges`，也会自动检测更改。

获取和设置未映射的属性的当前值

还可以读取未映射到数据库的属性的当前值。未映射的属性的一个示例可能是博客上的 `Rsslink` 属性。此值可以基于 `BlogId` 进行计算，因此无需存储在数据库中。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    // Read the current value of an unmapped property
    var rssLink = context.Entry(blog).Property(p => p.RssLink).CurrentValue;

    // Use a string to specify the property name
    var rssLinkAgain = context.Entry(blog).Property("RssLink").CurrentValue;
}
```

如果属性公开了 setter，还可以设置当前值。

在对未映射的属性执行实体框架验证时，读取未映射的属性的值非常有用。出于相同原因，可以读取当前值并将其设置为当前未由上下文跟踪的实体的属性。例如：

```
using (var context = new BloggingContext())
{
    // Create an entity that is not being tracked
    var blog = new Blog { Name = "ADO.NET Blog" };

    // Read and set the current value of Name as before
    var currentName1 = context.Entry(blog).Property(u => u.Name).CurrentValue;
    context.Entry(blog).Property(u => u.Name).CurrentValue = "My Fancy Blog";
    var currentName2 = context.Entry(blog).Property("Name").CurrentValue;
    context.Entry(blog).Property("Name").CurrentValue = "My Boring Blog";
}
```

请注意，原始值不可用于未映射的属性或上下文未跟踪的实体属性。

检查属性是否被标记为已修改

下面的示例演示如何检查单个属性是否被标记为已修改：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var nameIsModified1 = context.Entry(blog).Property(u => u.Name).IsModified;

    // Use a string for the property name
    var nameIsModified2 = context.Entry(blog).Property("Name").IsModified;
}
```

调用 SaveChanges 时，已修改属性的值将作为更新发送到数据库。

将属性标记为已修改

下面的示例演示如何强制将单个属性标记为已修改：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    context.Entry(blog).Property(u => u.Name).IsModified = true;

    // Use a string for the property name
    context.Entry(blog).Property("Name").IsModified = true;
}
```

将属性标记为已修改会强制在调用 SaveChanges 时将更新发送到该属性的数据库，即使该属性的当前值与原始值相同也是如此。

目前不能在标记为 "已修改" 后将单个属性重置为不修改。这是我们计划在未来版本中提供支持的内容。

读取实体的所有属性的当前值、原始值和数据库值

下面的示例演示如何在数据库中读取实体的所有映射属性的当前值、原始值和值。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    // Make a modification to Name in the tracked entity
    blog.Name = "My Cool Blog";

    // Make a modification to Name in the database
    context.Database.SqlCommand("update dbo.Blogs set Name = 'My Boring Blog' where Id = 1");

    // Print out current, original, and database values
    Console.WriteLine("Current values:");
    PrintValues(context.Entry(blog).CurrentValues);

    Console.WriteLine("\nOriginal values:");
    PrintValues(context.Entry(blog).OriginalValues);

    Console.WriteLine("\nDatabase values:");
    PrintValues(context.Entry(blog).GetDatabaseValues());
}

public static void PrintValues(DbPropertyValues values)
{
    foreach (var propertyName in values.PropertyNames)
    {
        Console.WriteLine("Property {0} has value {1}",
                          propertyName, values[propertyName]);
    }
}
```

当前值是实体的属性当前包含的值。原始值是查询实体时从数据库中读取的值。数据库值是当前存储在数据库中的值。如果数据库中的值可能在查询实体后发生更改（例如，当另一个用户对数据库进行了并发编辑时），获取数据库值将很有用。

设置其他对象的当前值或原始值

可以通过从另一个对象复制值来更新已跟踪实体的当前值或原始值。例如：

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    var coolBlog = new Blog { Id = 1, Name = "My Cool Blog" };
    var boringBlog = new BlogDto { Id = 1, Name = "My Boring Blog" };

    // Change the current and original values by copying the values from other objects
    var entry = context.Entry(blog);
    entry.CurrentValues.SetValues(coolBlog);
    entry.OriginalValues.SetValues(boringBlog);

    // Print out current and original values
    Console.WriteLine("Current values:");
    PrintValues(entry.CurrentValues);

    Console.WriteLine("\nOriginal values:");
    PrintValues(entry.OriginalValues);
}

public class BlogDto
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

运行上面的代码将输出：

```

Current values:
Property Id has value 1
Property Name has value My Cool Blog

Original values:
Property Id has value 1
Property Name has value My Boring Blog

```

当使用从服务调用或 n 层应用程序中的客户端获取的值更新实体时，有时会使用此方法。请注意，所使用的对象不必与实体具有相同的类型，但前提是它具有与实体的名称相匹配的属性。在上面的示例中，BlogDTO 的实例用于更新原始值。

请注意，从其他对象复制时，仅将设置为不同值的属性标记为已修改。

从字典设置当前值或原始值

可以通过从字典或其他一些数据结构复制值来更新已跟踪实体的当前值或原始值。例如：

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var newValues = new Dictionary<string, object>
    {
        { "Name", "The New ADO.NET Blog" },
        { "Url", "blogs.msdn.com/adonet" },
    };

    var currentValues = context.Entry(blog).CurrentValues;

    foreach (var propertyName in newValues.Keys)
    {
        currentValues[propertyName] = newValues[propertyName];
    }

    PrintValues(currentValues);
}

```

使用 OriginalValues 属性而非 CurrentValues 属性来设置原始值。

使用属性设置字典中的当前值或原始值

如上所述，使用 CurrentValues 或 OriginalValues 的一种替代方法是使用属性方法来设置每个属性的值。当你需要设置复杂属性的值时，这可能更好。例如：

```

using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    var newValues = new Dictionary<string, object>
    {
        { "Name", "John Doe" },
        { "Location.City", "Redmond" },
        { "Location.State.Name", "Washington" },
        { "Location.State.Code", "WA" },
    };

    var entry = context.Entry(user);

    foreach (var propertyName in newValues.Keys)
    {
        entry.Property(propertyName).CurrentValue = newValues[propertyName];
    }
}

```

在上面的示例中，使用点分名称访问复杂属性。有关访问复杂属性的其他方法，请参阅本主题后面有关复杂属性的两个部分。

创建包含当前、原始或数据库值的克隆对象

从 CurrentValues、OriginalValues 或 GetDatabaseValues 返回的 DbPropertyValues 对象可用于创建实体的克隆。此克隆将包含 DbPropertyValues 对象中用于创建它的属性值。例如：

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);

    var clonedBlog = context.Entry(blog).GetDatabaseValues().ToObject();
}

```

请注意，返回的对象不是实体，也不是由上下文跟踪。返回的对象也不会将任何关系设置为其他对象。

克隆的对象可用于解决与数据库并发更新相关的问题，尤其是在使用涉及到特定类型的对象的数据绑定的 UI 时。

获取和设置复杂属性的当前值或原始值

可以使用属性方法读取和设置整个复杂对象的值，就像它可用于基元属性一样。此外，还可以向下钻取到复杂对象，并读取或设置该对象的属性，甚至是嵌套的对象。下面是一些示例：

```

using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    // Get the Location complex object
    var location = context.Entry(user)
        .Property(u => u.Location)
        .CurrentValue;

    // Get the nested State complex object using chained calls
    var state1 = context.Entry(user)
        .ComplexProperty(u => u.Location)
        .Property(l => l.State)
        .CurrentValue;

    // Get the nested State complex object using a single lambda expression
    var state2 = context.Entry(user)
        .Property(u => u.Location.State)
        .CurrentValue;

    // Get the nested State complex object using a dotted string
    var state3 = context.Entry(user)
        .Property("Location.State")
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using chained calls
    var name1 = context.Entry(user)
        .ComplexProperty(u => u.Location)
        .ComplexProperty(l => l.State)
        .Property(s => s.Name)
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using a single lambda expression
    var name2 = context.Entry(user)
        .Property(u => u.Location.State.Name)
        .CurrentValue;

    // Get the value of the Name property on the nested State complex object using a dotted string
    var name3 = context.Entry(user)
        .Property("Location.State.Name")
        .CurrentValue;
}

```

使用 `OriginalValue` 属性而非 `CurrentValue` 属性来获取或设置原始值。

请注意，可以使用属性或 `ComplexProperty` 方法来访问复杂属性。但是，如果想要使用其他属性或 `ComplexProperty` 调用深化到复杂对象，则必须使用 `ComplexProperty` 方法。

使用 DbPropertyValues 访问复杂属性

使用 CurrentValues、OriginalValues 或 GetDatabaseValues 获取实体的所有当前值、原始值或数据库值时，任何复杂属性的值将作为嵌套 DbPropertyValues 对象返回。然后，可以使用这些嵌套对象获取复杂对象的值。例如，以下方法将打印出所有属性的值，包括任何复杂属性的值和嵌套的复杂属性。

```
public static void WritePropertyValues(string parentPropertyName, DbPropertyValues propertyValues)
{
    foreach (var propertyName in propertyValues.PropertyNames)
    {
        var nestedValues = propertyValues[propertyName] as DbPropertyValues;
        if (nestedValues != null)
        {
            WritePropertyValues(parentPropertyName + propertyName + ".", nestedValues);
        }
        else
        {
            Console.WriteLine("Property {0}{1} has value {2}",
                parentPropertyName, propertyName,
                propertyValues[propertyName]);
        }
    }
}
```

若要打印出所有当前属性值，将调用方法，如下所示：

```
using (var context = new BloggingContext())
{
    var user = context.Users.Find("johndoe1987");

    WritePropertyValues("", context.Entry(user).CurrentValues);
}
```

处理并发冲突

2020/3/11 •

乐观并发性涉及到乐观地尝试将实体保存到数据库，希望数据在加载实体后未发生更改。如果事实证明数据已更改，则会引发异常，并且在尝试再次保存之前必须解决冲突。本主题介绍如何在实体框架中处理此类异常。本主题所介绍的方法同样适用于查询使用 Code First 和 EF 设计器创建的模型。

这篇文章并不适合完整讨论开放式并发。以下各节介绍了并发解决方案的一些知识，并显示了常见任务的模式。

其中的许多模式使用[属性值](#)中讨论的主题。

在使用独立关联(其中外键未映射到实体中的属性)时解决并发性问题比使用外键关联更难。因此，如果你要在应用程序中执行并发解析，则建议你始终将外键映射到你的实体中。以下所有示例假设你使用的是外键关联。

当尝试保存使用外键关联的实体时，如果检测到乐观并发异常，则 SaveChanges 将引发 DbUpdateConcurrencyException。

通过重载解决开放式并发异常(数据库入选)

可以使用 Reload.sql 方法，用数据库中的值覆盖当前实体的值。然后，通常以某种形式向用户返回该实体，并且这些实体必须重试更改，然后重新保存。例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;

        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Update the values of the entity that failed to save from the store
            ex.Entries.Single().Reload();
        }
    } while (saveFailed);
}
```

模拟并发异常的一种好方法是在 SaveChanges 调用上设置断点，然后使用其他工具(如 SQL Management Studio)修改要保存在数据库中的实体。您还可以在 SaveChanges 之前插入一行，以便使用 SqlCommand 直接更新数据库。例如：

```
context.Database.SqlCommand(
    "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
```

DbUpdateConcurrencyException 上的条目方法返回未能更新的实体的 DbSet 实例。(此属性当前总是返回

并发问题的单个值。对于常规更新异常，它可能会返回多个值。)在某些情况下，另一种方法可能是从数据库中获取可能需要重新加载的所有实体的条目，并为每个实体调用 "重新加载"。

将开放式并发异常解析为客户端入选

以上使用重载的示例有时被称为数据库入选或存储入选，因为该实体中的值由数据库中的值覆盖。有时，您可能希望执行相反的操作，并用实体中当前的值覆盖数据库中的值。这有时称为客户端入选，可以通过获取当前数据库值并将其设置为实体的原始值来完成。(有关当前值和原始值的信息，请参阅使用[属性值](#)。)例如：

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Update original values from the database
            var entry = ex.Entries.Single();
            entry.OriginalValues.SetValues(entry.GetDatabaseValues());
        }
    } while (saveFailed);
}
```

开放式并发异常的自定义解决

有时，您可能想要将数据库中当前的值与实体中的当前值组合在一起。这通常需要一些自定义逻辑或用户交互。例如，您可能向用户提供窗体，其中包含当前值、数据库中的值和一组默认的已解析值。然后，用户将根据需要编辑解析的值，并将这些已解决的值保存到数据库中。为此，可以使用 `DbPropertyValues` 对象，该对象是从实体条目的 `CurrentValues` 和 `GetDatabaseValues` 返回的。例如：

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Get the current entity values and the values in the database
            var entry = ex.Entries.Single();
            var currentValue = entry.CurrentValues;
            var databaseValues = entry.GetDatabaseValues();

            // Choose an initial set of resolved values. In this case we
            // make the default be the values currently in the database.
            var resolvedValues = databaseValues.Clone();

            // Have the user choose what the resolved values should be
            HaveUserResolveConcurrency(currentValue, databaseValues, resolvedValues);

            // Update the original values with the database values and
            // the current values with whatever the user choose.
            entry.OriginalValues.SetValues(databaseValues);
            entry.CurrentValues.SetValues(resolvedValues);
        }
    } while (saveFailed);
}

public void HaveUserResolveConcurrency(DbPropertyValues currentValue,
                                       DbPropertyValues databaseValues,
                                       DbPropertyValues resolvedValues)
{
    // Show the current, database, and resolved values to the user and have
    // them edit the resolved values to get the correct resolution.
}

```

使用对象的开放式并发异常的自定义解决

上面的代码使用 `DbPropertyValues` 实例来传递当前、数据库和解析的值。有时，使用实体类型的实例可能会更容易。可以使用 `DbPropertyValues` 的 `ToObject` 和 `SetValues` 方法完成此操作。例如：

```

using (var context = new BloggingContext())
{
    var blog = context.Blogs.Find(1);
    blog.Name = "The New ADO.NET Blog";

    bool saveFailed;
    do
    {
        saveFailed = false;
        try
        {
            context.SaveChanges();
        }
        catch (DbUpdateConcurrencyException ex)
        {
            saveFailed = true;

            // Get the current entity values and the values in the database
            // as instances of the entity type
            var entry = ex.Entries.Single();
            var databaseValues = entry.GetDatabaseValues();
            var databaseValuesAsBlog = (Blog)databaseValues.ToObject();

            // Choose an initial set of resolved values. In this case we
            // make the default be the values currently in the database.
            var resolvedValuesAsBlog = (Blog)databaseValues.ToObject();

            // Have the user choose what the resolved values should be
            HaveUserResolveConcurrency((Blog)entry.Entity,
                databaseValuesAsBlog,
                resolvedValuesAsBlog);

            // Update the original values with the database values and
            // the current values with whatever the user choose.
            entry.OriginalValues.SetValues(databaseValues);
            entry.CurrentValues.SetValues(resolvedValuesAsBlog);
        }
    } while (saveFailed);
}

public void HaveUserResolveConcurrency(Blog entity,
    Blog databaseValues,
    Blog resolvedValues)
{
    // Show the current, database, and resolved values to the user and have
    // them update the resolved values to get the correct resolution.
}

```

使用事务

2020/3/11 •

NOTE

■ EF6 ■ - 此页面中讨论的功能、API 等已引入实体框架 6。如果使用的是早期版本，则部分或全部信息不适用。

本文档介绍如何在 EF6 中使用事务，包括自 EF5 后添加的增强功能，以便轻松处理事务。

默认情况下，什么是 EF

在所有版本的实体框架中，每当你执行 `SaveChanges ()` 在数据库中插入、更新或删除操作时，框架会将该操作包装在事务中。此事务只持续足够长的时间来执行操作，然后完成。执行另一个这样的操作时，将启动新事务。

从 EF6 开始，`ExecuteSqlCommand ()` 在默认情况下会在事务中包装命令（如果尚未存在）。此方法有一些重载，允许你根据需要重写此行为。此外，通过 `ExecuteFunction ()` 等 API，在模型中包含的存储过程的执行也是相同的（但在重写默认行为时不能 EF6）。

在任一情况下，事务的隔离级别都是数据库提供程序认为其默认设置的任何隔离级别。例如，默认情况下，在 SQL Server 此为“已提交读”。

实体框架不会在事务中包装查询。

此默认功能适用于很多用户，如果没有，则无需在 EF6 中执行任何其他操作；只需像往常一样编写代码。

但是，某些用户需要更好地控制其事务—以下各节将对此进行介绍。

API 的工作原理

在实体框架 EF6 之前，请打开数据库连接本身（如果传递了已打开的连接，则会引发异常）。由于只能在打开的连接上启动事务，这意味着用户可以将多个操作包装到一个事务中的唯一方法是使用 `TransactionScope` 或使用 `ObjectContext` 属性，并直接对返回的 `EntityConnection` 对象调用 `open ()` 和 `BeginTransaction ()`。此外，如果你在基础数据库连接上自行启动了事务，则与数据库联系的 API 调用将失败。

NOTE

实体框架 6 中删除了仅接受关闭的连接的限制。有关详细信息，请参阅 [连接管理](#)。

从 EF6 开始，框架现在提供：

1. `BeginTransaction ()`：一种更简单的方法，使用户能够在现有的 `DbContext` 内自行启动和完成事务—允许将多个操作合并到同一个事务中，并因此全部提交或全部回滚。它还允许用户更轻松地指定事务的隔离级别。
2. `UseTransaction ()`：这允许 `DbContext` 使用在实体框架之外启动的事务。

将多个操作合并为同一上下文中的一个事务

`BeginTransaction ()` 具有两个替代，其中一个将使用显式 `IsolationLevel`，另一个不采用任何参数，并从基础数据库提供程序使用默认的 `IsolationLevel`。这两个重写都返回一个处理对象，该对象提供对基础存储区事务执行 `commit` 和 `Rollback` 的 `commit ()` 和 `rollback ()` 方法。

处理在提交或回滚后会被释放。实现此目的的一种简单方法是使用 `(...){...}` 当使用块完成时，将自动调用 `Dispose ()` 的语法：

```

using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void StartOwnTransactionWithinContext()
        {
            using (var context = new BloggingContext())
            {
                using (var dbContextTransaction = context.Database.BeginTransaction())
                {
                    context.Database.ExecuteSqlCommand(
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'"
                    );

                    var query = context.Posts.Where(p => p.Blog.Rating >= 5);
                    foreach (var post in query)
                    {
                        post.Title += "[Cool Blog]";
                    }

                    context.SaveChanges();

                    dbContextTransaction.Commit();
                }
            }
        }
    }
}

```

NOTE

开始事务要求基础存储连接处于打开状态。因此调用 `BeginTransaction()` 将打开连接（如果尚未打开）。如果处理打开了连接，则在调用 `Dispose()` 时，它将关闭该连接。

将现有事务传递到上下文

有时，您想要在范围内更广泛的事务，其中包括对同一数据库的操作，而不是在 EF 的外部。若要实现此目的，您必须打开连接并自行启动事务，然后告诉 EF a) 以使用已打开的数据库连接，使用 b) 在该连接上使用现有事务。

若要执行此操作，必须在上下文类上定义和使用一个构造函数，该构造函数继承自一个 `DbContext` 构造函数，该构造函数使用 i) 一个 `contextOwnsConnection` 布尔值。

NOTE

在这种情况下，`contextOwnsConnection` 标志必须设置为 `false`。这一点很重要，因为它会通知实体框架它不应在处理完成后关闭连接（例如，请参阅下面的第4行）：

```
using (var conn = new SqlConnection("..."))
{
    conn.Open();
    using (var context = new BloggingContext(conn, contextOwnsConnection: false))
    {
    }
}
```

而且，您必须自行启动事务（如果您想要避免默认设置，则包括 `IsolationLevel`），并让实体框架知道已经在连接上启动了现有事务（请参阅下面的第33行）。

然后，可以随意直接在 `SqlConnection` 本身或 `DbContext` 上执行数据库操作。所有此类操作在一个事务内执行。您需要负责提交或回滚事务，并对其调用 `Dispose()`，以及关闭并释放数据库连接。例如：

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void UsingExternalTransaction()
        {
            using (var conn = new SqlConnection("..."))
            {
                conn.Open();

                using (var sqlTxn = conn.BeginTransaction(System.Data.IsolationLevel.Snapshot))
                {
                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
                    sqlCommand.Transaction = sqlTxn;
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'";
                    sqlCommand.ExecuteNonQuery();

                    using (var context =
                        new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        context.Database.UseTransaction(sqlTxn);

                        var query = context.Posts.Where(p => p.Blog.Rating >= 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }
                        context.SaveChanges();
                    }

                    sqlTxn.Commit();
                }
            }
        }
    }
}
```

清除事务

您可以将 `null` 传递给 `UseTransaction()` 以清除当前事务实体框架的知识。当你执行此操作时，实体框架不会提交

或回滚现有事务，因此请谨慎使用，仅在你确定要执行的操作时使用。

UseTransaction 中的错误

如果在以下情况传递事务，则会看到来自 UseTransaction () 的异常：

- 实体框架已有一个现有的事务
- 实体框架已在某一 TransactionScope 内运行
- 传递的事务中的连接对象为 null。也就是说，事务与连接无关-通常是该事务已完成的符号
- 传递的事务中的连接对象与实体框架的连接不匹配。

将事务与其他功能一起使用

本部分详细说明了上述事务与的交互方式：

- 连接复原
- 异步方法
- TransactionScope 交易

连接复原

新的连接复原功能不适用于用户启动的事务。有关详细信息，请参阅[重试执行策略](#)。

异步编程

前面几节中所述的方法不需要更多选项或设置即可使用[异步查询和保存方法](#)。但请注意，根据您在异步方法中所执行的操作，这可能会导致长时间运行的事务，进而导致死锁或阻塞，导致整个应用程序的性能不佳。

TransactionScope 交易

在 EF6 之前，提供更大范围事务的建议方法是使用 TransactionScope 对象：

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        static void UsingTransactionScope()
        {
            using (var scope = new TransactionScope(TransactionScopeOption.Required))
            {
                using (var conn = new SqlConnection("..."))
                {
                    conn.Open();

                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'";
                    sqlCommand.ExecuteNonQuery();

                    using (var context =
                        new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        var query = context.Posts.Where(p => p.Blog.Rating > 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }
                        context.SaveChanges();
                    }
                }

                scope.Complete();
            }
        }
    }
}

```

`SqlConnection` 和实体框架都使用环境 `TransactionScope` 事务，因此一起提交。

从 .NET 4.5.1 `TransactionScope` 开始，还更新为通过使用[TransactionScopeAsyncFlowOption](#)枚举来处理异步方法：

```

using System.Collections.Generic;
using System.Data.Entity;
using System.Data.SqlClient;
using System.Linq;
using System.Transactions;

namespace TransactionsExamples
{
    class TransactionsExample
    {
        public static void AsyncTransactionScope()
        {
            using (var scope = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
            {
                using (var conn = new SqlConnection("..."))
                {
                    await conn.OpenAsync();

                    var sqlCommand = new SqlCommand();
                    sqlCommand.Connection = conn;
                    sqlCommand.CommandText =
                        @"UPDATE Blogs SET Rating = 5" +
                        " WHERE Name LIKE '%Entity Framework%'";
                    await sqlCommand.ExecuteNonQueryAsync();

                    using (var context = new BloggingContext(conn, contextOwnsConnection: false))
                    {
                        var query = context.Posts.Where(p => p.Blog.Rating > 5);
                        foreach (var post in query)
                        {
                            post.Title += "[Cool Blog]";
                        }

                        await context.SaveChangesAsync();
                    }
                }
            }
        }
    }
}

```

对于 TransactionScope 方法仍有一些限制：

- 需要 .NET 4.5.1 或更高版本才能使用异步方法。
- 除非你确定有且只有一个连接（云方案不支持分布式事务），否则不能在云方案中使用它。
- 它不能与前面部分的 UseTransaction () 方法结合。
- 如果你颁发了任何 DDL 并且尚未通过 MSDTC 服务启用分布式事务，则它会引发异常。

TransactionScope 方法的优点：

- 如果与同一事务中的另一个数据库建立了多个连接，则它会自动将本地事务升级到分布式事务，或将连接合并到一个数据库以连接到同一事务中的另一个数据库（注意：您必须MSDTC 服务配置为允许此操作运行的分布式事务）。
- 易于编码。如果你更愿意在后台隐式处理事务，而不是显式地在控制下进行处理，则 TransactionScope 方法可能更适合你。

总而言之，对于上述新的 BeginTransaction () 和 UseTransaction () API，大多数用户不再需要 TransactionScope 方法。如果继续使用 TransactionScope，请注意上述限制。建议尽可能使用前面部分中所述的方法。

数据验证

2020/3/11 •

NOTE

■ ef 4.1 - 在实体框架4.1 中引入了本页中所述的功能、api 等。如果使用的是早期版本，则不会应用部分或全部信息。

此页面上的内容适用于最初由 Julie Lerman (<https://thedatafarm.com>) 编写的文章。

实体框架提供了一种非常丰富的验证功能，这些功能可用于向用户界面提供客户端验证或用于服务器端验证。使用 code first 时，可以使用批注或 Fluent API 配置来指定验证。可以在代码中指定其他验证和更复杂的验证，无论模型是从代码 hails、模型优先还是数据库优先进行操作，都可以使用。

模型

我将使用一对简单的类来演示验证：博客和文章。

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set; }
    public DateTime DateCreated { get; set; }
    public virtual ICollection<Post> Posts { get; set; }
}

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}
```

数据注释

Code First 使用 `System.ComponentModel.DataAnnotations` 程序集中的批注作为配置代码优先类的一种方法。这些批注中包括 `Required`、`MaxLength` 和 `MinLength` 等规则。许多 .NET 客户端应用程序还识别这些批注，例如，ASP.NET MVC。您可以通过这些批注实现客户端和服务器端验证。例如，你可以强制 "博客 Title" 属性为必需属性。

```
[Required]
public string Title { get; set; }
```

如果应用程序中没有额外的代码或标记更改，则现有 MVC 应用程序会执行客户端验证，甚至使用属性和批注名称动态生成消息。

Create

Blog

Title
 The Title field is required.

BloggerName
 Julie

DateCreated
 3/15/2011

在此“创建”视图的“回发”方法中，实体框架用于将新的博客保存到数据库中，但在应用程序到达该代码之前触发了 MVC 的客户端验证。

不过，客户端验证不是项目符号。用户可能会影响浏览器的功能或更糟的功能，黑客可能会使用某些 trickery 来避免 UI 验证。但实体框架还将识别 `Required` 注释并对其进行验证。

对此进行测试的一种简单方法是禁用 MVC 的客户端验证功能。可以在 MVC 应用程序的 web.config 文件中执行此操作。`AppSettings` 节具有 `ClientValidationEnabled` 的键。将此项设置为 `false` 将阻止 UI 执行验证。

```
<appSettings>
  <add key="ClientValidationEnabled" value="false"/>
  ...
</appSettings>
```

即使已禁用客户端验证，也会在应用程序中获得相同的响应。错误消息“需要标题字段”将像以前一样显示。除非现在是服务器端验证的结果。实体框架将在 `Required` 注释上执行验证（甚至在麻烦之前生成用于发送到数据库的 `INSERT` 命令），并将错误返回到将显示消息的 MVC。

Fluent API

您可以使用 code first 的 Fluent API 而不是使用批注来获取相同的客户端 & 服务器端验证。我将使用 `MaxLength` 验证来说明这一点，而不是使用 `Required`。

当代码优先从类生成模型时，将应用“熟知 API 配置”。可以通过重写 `DbContext` 类的 `OnModelCreating` 方法注入配置。以下配置指定 `BloggerName` 属性的长度不能超过10个字符。

```
public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>().Property(p => p.BloggerName).HasMaxLength(10);
    }
}
```

基于流畅 API 配置引发的验证错误不会自动到达 UI，但你可以在代码中捕获该错误，然后相应地对其进行响应。

下面是应用程序的 `BlogController` 类中的一些异常处理错误代码，该代码在实体框架尝试使用超过10个字符的 `BloggerName` 保存博客时捕获验证错误。

```
[HttpPost]
public ActionResult Edit(int id, Blog blog)
{
    try
    {
        db.Entry(blog).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch (DbEntityValidationException ex)
    {
        var error = ex.EntityValidationErrors.First().ValidationErrors.First();
        this.ModelState.AddModelError(error.PropertyName, error.ErrorMessage);
        return View();
    }
}
```

验证不会自动传递回视图，这就是使用 `ModelState.AddModelError` 的其他代码的原因。这可以确保错误详细信息使其成为视图，然后将使用 `ValidationMessageFor` `HtmlHelper` 显示错误。

```
@Html.ValidationMessageFor(model => model.BloggerName)
```

IValidatableObject

`IValidatableObject` 是 `System.ComponentModel.DataAnnotations` 的接口。尽管它不是实体框架 API 的一部分，但仍可在实体框架类中利用它进行服务器端验证。`IValidatableObject` 提供实体框架将在 `SaveChanges` 期间调用的 `Validate` 方法，你可以随时调用以验证类。

`Required` 和 `MaxLength` 等配置对单个字段执行验证。在 `Validate` 方法中，可以使用更复杂的逻辑，例如，比较两个字段。

在下面的示例中，扩展了 `Blog` 类以实现 `IValidatableObject`，然后提供 `Title` 和 `BloggerName` 不能匹配的规则。

```
public class Blog : IValidatableObject
{
    public int Id { get; set; }

    [Required]
    public string Title { get; set; }

    public string BloggerName { get; set; }
    public DateTime DateCreated { get; set; }
    public virtual ICollection<Post> Posts { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (Title == BloggerName)
        {
            yield return new ValidationResult(
                "Blog Title cannot match Blogger Name",
                new[] { nameof(Title), nameof(BloggerName) });
        }
    }
}
```

`ValidationResult` 构造函数采用一个表示错误消息的 `string`，以及一个 `string` 的数组，该数组表示与验证关联的成员名称。由于此验证同时检查 `Title` 和 `BloggerName`，因此将返回这两个属性名。

与“流畅 API”提供的验证不同，此验证结果将被视图识别，并且之前用于将错误添加到 `ModelState` 中的异常处理

程序不是必需的。因为我在 `ValidationResult` 中设置了两个属性名，所以 MVC HtmlHelpers 将显示这两个属性的错误消息。

Edit

Blog

Title
 Blog Title cannot match Blogger Name

BloggerName
 Blog Title cannot match Blogger Name

DateCreated

DbContext.ValidateEntity

`DbContext` 具有称为 `ValidateEntity` 的可重写方法。调用 `SaveChanges` 时，实体框架将为其缓存中其状态为 "不 `Unchanged`" 的每个实体调用此方法。您可以在此处直接放置验证逻辑，甚至可以使用此方法来调用，例如，在上一节中添加的 `Blog.Validate` 方法。

下面是一个用于验证新的 `Post` 的 `ValidateEntity` 重写示例，以确保尚未使用帖子标题。它首先检查实体是否为 `post` 并添加其状态。如果是这种情况，则会在数据库中查找具有相同标题的帖子。如果已存在现有的公告，则会创建一个新的 `DbEntityValidationResult`。

`DbEntityValidationResult` 承载单个实体的 `DbEntityEntry` 和 `IEnumerable<DbValidationError>`。此方法开始时，实例化 `DbEntityValidationResult`，然后将发现的任何错误添加到其 `ValidationErrors` 集合。

```
protected override DbEntityValidationResult ValidateEntity (
    System.Data.Entity.Infrastructure.DbEntityEntry entityEntry,
    IDictionary<object, object> items)
{
    var result = new DbEntityValidationResult(entityEntry, new List<DbValidationError>());

    if (entityEntry.Entity is Post post && entityEntry.State == EntityState.Added)
    {
        // Check for uniqueness of post title
        if (Posts.Where(p => p.Title == post.Title).Any())
        {
            result.ValidationErrors.Add(
                new System.Data.Entity.Validation.DbValidationError(
                    nameof(Post.Title),
                    "Post title must be unique."));
        }
    }

    if (result.ValidationErrors.Count > 0)
    {
        return result;
    }
    else
    {
        return base.ValidateEntity(entityEntry, items);
    }
}
```

显式触发验证

对 `SaveChanges` 的调用会触发本文中所述的所有验证。但不需要依赖 `SaveChanges`。你可能希望在应用程序中的其他地方进行验证。

`DbContext.GetValidationErrors` 将触发所有验证、批注定义的验证或流畅的 API、在 `IValidatableObject` 中创建的验证(例如 `Blog.Validate`)以及在 `DbContext.ValidateEntity` 方法中执行的验证。

下面的代码将对 `DbContext` 的当前实例调用 `GetValidationErrors`。`ValidationErrors` 按实体类型分组到 `DbEntityValidationResult` 中。此代码首先遍历方法返回的 `DbEntityValidationResult`，然后遍历其中的每个 `DbValidationError`。

```
foreach (var validationResult in db.GetValidationErrors())
{
    foreach (var error in validationResult.ValidationErrors)
    {
        Debug.WriteLine(
            "Entity Property: {0}, Error {1}",
            error.PropertyName,
            error.ErrorMessage);
    }
}
```

使用验证时的其他注意事项

下面是在使用实体框架验证时要考虑的一些其他事项：

- 在验证过程中禁用延迟加载
- EF 将验证非映射属性(未映射到数据库中的列的属性)中的数据批注
- 在 `SaveChanges` 期间检测到更改后，将执行验证。如果在验证期间进行更改，则需要负责通知更改跟踪器
- 如果在验证期间发生错误，则会引发 `DbUnexpectedValidationException`
- 实体框架包括在模型中的方面(最大长度、必需等)将导致验证，即使类和/或使用 EF 设计器创建模型
- 优先规则：
 - 流畅的 API 调用替代相应的数据批注
- 执行顺序：
 - 属性验证发生在类型验证之前
 - 仅当属性验证成功时才发生类型验证
- 如果某个属性是复杂的，则其验证也包括：
 - 针对复杂类型属性的属性级验证
 - 对复杂类型的类型级别验证，包括对复杂类型 `IValidatableObject` 验证

摘要

实体框架中的验证 API 与 MVC 中的客户端验证非常完美，但你不必依赖于客户端验证。实体框架将在服务器端对 DataAnnotations 或使用 code first 流畅 API 应用的配置进行验证。

你还看到了许多扩展点，用于自定义该行为，无论你使用 `IValidatableObject` 接口还是点击 `DbContext.ValidateEntity` 方法。无论是使用 Code First、Model First 还是 Database First 工作流来描述概念模型，最后两种验证方法都可通过 `DbContext` 来使用。

实体框架博客

2020/3/11 •

除了产品文档，这些博客可能是有关实体框架的有用信息的源：

EF 团队博客

- [.NET 博客-标记: 实体框架](#)
- [ADO.NET 博客\(不再使用\)](#)
- [EF 设计博客\(不再使用\)](#)

当前和以前的 EF 团队博主

- [Arthur Vickers](#)
- [Brice Lambson](#)
- [圣地亚哥 Vega](#)
- [Rowan Miller](#)
- [Pawel Kadluczka](#)
- [Alex James](#)
- [Zlatko Michailov](#)

EF 社区博客

- [Julie Lerman](#)
- [Shawn Wildermuth](#)

实体框架的 Microsoft 案例研究

2020/3/11 •

此页上的案例研究重点介绍了几个实体框架的实际生产项目。

NOTE

这些案例研究的详细版本在 Microsoft 网站上不再可用。因此，这些链接已被删除。

Epicor

Epicor 是一家大型的全球软件公司(超过400个开发人员)，可为超过150个国家/地区的公司开发企业资源规划(ERP)解决方案。其旗舰版产品 Epicor 9 基于面向服务的体系结构(SOA)，该体系结构使用 .NET Framework。面对大量客户请求以提供对语言集成查询(LINQ)的支持，并希望在其后端 SQL Server 上减少负载，团队决定升级到 Visual Studio 2010 和 .NET Framework 4.0。使用实体框架4.0，它们能够实现这些目标，同时大大简化了开发和维护工作。具体而言，实体框架的丰富 T4 支持使其能够完全控制其生成的代码，并自动生成性能保存功能，如预编译查询和缓存。

"我们最近使用现有代码执行了一些性能测试，并可以减少 90% SQL Server 的请求数。这是因为 ADO.NET 实体框架 4." – Erik Johnson, 副总裁, 产品研究

精确性解决方案

收购了一个事件规划软件系统，该系统会很难维护和扩展精确性的长期解决方案，使用 Visual Studio 2010 将其重新编写为内置于 Silverlight 4 的功能强大且易于使用的丰富 Internet 应用程序。使用 .NET RIA 服务，他们能够在实体框架上快速构建一个服务层，以避免在不同的层对常见验证和身份验证逻辑进行代码重复和允许。

"我们在第一次推出时实体框架销售，而实体框架4已证实更好。工具经过改进，可以更轻松地操作定义概念模型、存储模型和这些模型之间的映射的.edmx 文件。使用实体框架，我可以将数据访问层作为一天的工作，并在我开始时进行构建。实体框架是我们事实上的数据访问层；我不知道什么人不会使用它。" – Joe McBride, 资深开发人员

北美洲显示的 NEC 解决方案

NEC 需要为基于数字位置的广告输入市场，并提供一种解决方案，让广告商和网络所有者受益并增加自己的收入。为此，它会启动一对 web 应用程序，以自动执行传统 ad 活动中需要的手动过程。这些站点是使用 ASP.NET、Silverlight 3、AJAX 和 WCF 生成的，以及在数据访问层中的实体框架与 SQL Server 2008 进行通信。

"借助 SQL Server，我们认为我们可以获得所需的吞吐量，以提供实时信息和可靠性来帮助广告商和网络，从而帮助确保关键任务应用程序中的信息始终可用"-Mike CorcoranIT 主管

达尔文维度

使用范围广泛的 Microsoft 技术，达尔文的团队设置为创建 Evolver—一个在线虚拟形象门户，使用者可以使用它创建令人惊叹的、逼真的头像，以便在游戏、动画和社交网络页面上使用。随着实体框架的工作效率优势，并请求 Windows Workflow Foundation (WF) 和 Windows Server AppFabric (一种高度可缩放的内存中应用程序缓存) 等组件，团队能够在35% 的时间内提供令人惊叹的产品开发时间。即使团队成员拆分到多个国家/地区，团队仍遵循每周发布的敏捷开发流程。

"我们不会尝试创建技术来实现技术。作为一种启动方式，我们需要利用可节省时间和资金的技术，这一点非常重要。.NET 是用于快速、经济高效的开发的选择。 - Zachary Olsen, 建筑师

Silverware

由于为中小型餐厅组开发销售销售点(POS)解决方案的经验超过了15年, Silverware 的开发团队将其产品设置为通过更多的企业级功能来增强其产品, 以吸引更大餐馆链。使用最新版本的 Microsoft 开发工具, 能够比以前更快地构建新解决方案。LINQ 和实体框架等关键新功能使得从 Crystal 报表移到 SQL Server 2008 和 SQL Server Reporting Services (SSRS), 以实现其数据存储和报告需求。

"有效的数据管理是 SilverWare 成功的关键, 这就是我们决定采用 SQL Reporting 的原因。" -IT/软件工程主管 Nicholas Romanidis

参与实体框架6

2020/3/11 •

使用 GitHub 上的开源模型开发实体框架6。尽管 Microsoft 的实体框架团队的主要重点在于向 Entity Framework Core 中添加新功能，但我们不想将任何主要功能添加到实体框架6，但我们仍接受贡献。

对于产品发布，请从[GitHub 存储库中的参与 wiki 页面](#)开始。

若要查看文档，请阅读[文档存储库中的贡献指南](#)。

使用实体框架获取帮助

2020/3/11 ·



有关使用 EF 的问题

使用实体框架获取帮助的最佳方式是使用[Entity Framework](#)标记在 Stack Overflow 上发布问题。

如果您对 Stack Overflow 不熟悉, 请务必[阅读询问问题的指南](#)。特别是, 不要使用 Stack Overflow 来报告 bug、提出路线图问题或建议新功能。



Bug 报告和功能请求

如果找到了应修复的 bug, 请使用要查看的功能, 或找不到答案的问题, 请在[EF6 GitHub 存储库](#)中创建问题。

实体框架术语表

2020/3/16 •

Code First

使用代码创建实体框架模型。该模型可以面向现有数据库或新数据库。

上下文

一个类，它表示与数据库的会话，使您能够查询和保存数据。上下文派生自 DbContext 或 ObjectContext 类。

约定 (Code First)

实体框架使用来从类推断模型形状的规则。

Database First

使用 EF 设计器创建面向现有数据库的实体框架模型。

预先加载

加载相关数据的一种模式，其中一种类型的实体的查询还会将相关实体作为查询的一部分进行加载。

EF 设计器

Visual Studio 中的可视化设计器，可用于使用框和线条创建实体框架模型。

实体

表示客户、产品和订单等应用程序数据的类或对象。

实体数据模型

描述实体和它们之间的关系的模型。EF 使用 EDM 来描述开发人员计划的概念模型。EDM 在 Chen 引入的实体关系模型的基础上生成。EDM 最初是使用成为 Microsoft 的一系列开发人员和服务器技术的主要目标而开发的。EDM 还可作为 OData 协议的一部分。

显式加载

一种用于加载相关数据的模式，其中通过调用 API 加载相关的对象。

Fluent API

可用于配置 Code First 模型的 API。

外键关联

实体之间的关联，其中表示外键的属性包含在依赖实体的类中。例如，Product 包含 "类别 Id" 属性。

标识关系

一种关系，其中主体实体的主键是依赖实体的主键的一部分。在这种关系中，没有主体实体，依赖实体就不能存在。

独立关联

实体之间的关联，其中不存在表示依赖实体的类中的外键的属性。例如，Product 类包含与 Category 的关系，但没有 "类别 Id" 属性。实体框架跟踪关联的状态，这与两个关联端的实体状态无关。

延迟加载

一种加载相关数据的模式，在访问导航属性时，会自动加载相关的对象。

Model First

使用 EF 设计器创建实体框架模型，然后使用该设计器创建新的数据库。

导航属性

引用其他实体的实体的属性。例如，Product 包含类别导航属性，类别包含 Products 导航属性。

POCO

纯旧式 CLR 对象的缩写。不与任何框架具有依赖关系的简单用户类。在 EF 的上下文中，不是派生自 EntityObject 的实体类实现任何接口，或携带 EF 中定义的任何属性。与持久性框架分离的此类实体类也称为 "持久性未知"。

逆关系

关系的相反结束，例如 product。类别和类别。产品.

自跟踪实体

从代码生成模板生成的实体，可帮助进行 N 层开发。

每个具体的表类型(TPC)

一种映射继承的方法，其中层次结构中的每个非抽象类型都映射到数据库中的单独表。

每个层次结构一个表(TPH)

一种映射继承的方法，其中，层次结构中的所有类型都映射到数据库中的同一表。鉴别器列用于标识与每行关联的类型。

每种类型一个表(TPT)

一种映射继承的方法，其中层次结构中所有类型的公共属性都映射到数据库中的同一个表，但每个类型独有的属性将映射到一个单独的表。

类型发现

标识应属于实体框架模型的类型的过程。

School 示例数据库

2020/3/11 •

本主题包含 School 数据库的架构和数据。示例 School 数据库在整个实体框架文档的不同位置使用。

NOTE

随 Visual Studio 一起安装的数据库服务器取决于你使用的 Visual Studio 版本。有关使用内容的详细信息, 请参阅[Visual Studio 版本](#)。

下面是创建数据库的步骤:

- 打开 Visual Studio
- 查看 -> 服务器资源管理器
- 右键单击 "数据连接" -> "添加连接 ... "
- 如果尚未从服务器资源管理器连接到数据库, 则需要选择Microsoft SQL Server作为数据源
- 连接到 LocalDB 或 SQL Express, 具体取决于你安装的是哪个
- 输入School作为数据库名称
- 选择 "确定", 系统会询问您是否要创建新数据库, 请选择 "是"
- 新数据库现在将出现在服务器资源管理器
- 如果使用的是 Visual Studio 2012 或更高版本
 - 在服务器资源管理器中右键单击该数据库, 然后选择 "新建查询"
 - 将以下 SQL 复制到新的查询中, 然后右键单击该查询, 然后选择 "执行"
- 如果使用的是 Visual Studio 2010
 - 选择数据 -> Transact-sql 编辑器 -> 新建查询连接 ...
 - 输入 .\SQLEXPRESS 作为服务器名称, 然后单击 "确定"
 - 从 "查询编辑器" 顶部的下拉菜单中选择 "STESample" 数据库
 - 将以下 SQL 复制到新的查询中, 然后右键单击该查询, 然后选择 "执行 SQL "。

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

-- Create the Department table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Department]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Department]([DepartmentID] [int] NOT NULL,
[Name] [nvarchar](50) NOT NULL,
[Budget] [money] NOT NULL,
[StartDate] [datetime] NOT NULL,
[Administrator] [int] NULL,
CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
(
[DepartmentID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the Person table.
IF NOT EXISTS (SELECT * FROM sys.objects
```

```

-- Create the Person table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Person]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Person]([PersonID] [int] IDENTITY(1,1) NOT NULL,
[LastName] [nvarchar](50) NOT NULL,
[FirstName] [nvarchar](50) NOT NULL,
[HireDate] [datetime] NULL,
[EnrollmentDate] [datetime] NULL,
[Discriminator] [nvarchar](50) NOT NULL,
CONSTRAINT [PK_School.Student] PRIMARY KEY CLUSTERED
(
[PersonID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OnsiteCourse table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OnsiteCourse]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OnsiteCourse]([CourseID] [int] NOT NULL,
[Location] [nvarchar](50) NOT NULL,
[Days] [nvarchar](50) NOT NULL,
[Time] [smalldatetime] NOT NULL,
CONSTRAINT [PK_OnsiteCourse] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OnlineCourse table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OnlineCourse]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OnlineCourse]([CourseID] [int] NOT NULL,
[URL] [nvarchar](100) NOT NULL,
CONSTRAINT [PK_OnlineCourse] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

--Create the StudentGrade table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[StudentGrade]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[StudentGrade]([EnrollmentID] [int] IDENTITY(1,1) NOT NULL,
[CourseID] [int] NOT NULL,
[StudentID] [int] NOT NULL,
[Grade] [decimal](3, 2) NULL,
CONSTRAINT [PK_StudentGrade] PRIMARY KEY CLUSTERED
(
[EnrollmentID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the CourseInstructor table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[CourseInstructor]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[CourseInstructor]([CourseID] [int] NOT NULL,
[InstructorID] [int] NOT NULL,
CONSTRAINT [PK_CourseInstructor] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

```

```

[PersonID] [int] NOT NULL,
CONSTRAINT [PK_CourseInstructor] PRIMARY KEY CLUSTERED
(
[CourseID] ASC,
[PersonID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the Course table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[Course]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[Course]([CourseID] [int] NOT NULL,
[Title] [nvarchar](100) NOT NULL,
[Credits] [int] NOT NULL,
[DepartmentID] [int] NOT NULL,
CONSTRAINT [PK_School.Course] PRIMARY KEY CLUSTERED
(
[CourseID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Create the OfficeAssignment table.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[OfficeAssignment]')
AND type in (N'U'))
BEGIN
CREATE TABLE [dbo].[OfficeAssignment]([InstructorID] [int] NOT NULL,
[Location] [nvarchar](50) NOT NULL,
[Timestamp] [timestamp] NOT NULL,
CONSTRAINT [PK_OfficeAssignment] PRIMARY KEY CLUSTERED
(
[InstructorID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]) ON [PRIMARY]
END
GO

-- Define the relationship between OnsiteCourse and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OnsiteCourse_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[OnsiteCourse]'))
ALTER TABLE [dbo].[OnsiteCourse] WITH CHECK ADD
CONSTRAINT [FK_OnsiteCourse_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO
ALTER TABLE [dbo].[OnsiteCourse] CHECK
CONSTRAINT [FK_OnsiteCourse_Course]
GO

-- Define the relationship between OnlineCourse and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OnlineCourse_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[OnlineCourse]'))
ALTER TABLE [dbo].[OnlineCourse] WITH CHECK ADD
CONSTRAINT [FK_OnlineCourse_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO
ALTER TABLE [dbo].[OnlineCourse] CHECK
CONSTRAINT [FK_OnlineCourse_Course]
GO

-- Define the relationship between StudentGrade and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_StudentGrade_Course]')
AND parent_object_id = OBJECT_ID(N'[dbo].[StudentGrade]'))
ALTER TABLE [dbo].[StudentGrade] WITH CHECK ADD

```

```

CONSTRAINT [FK_StudentGrade_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO
ALTER TABLE [dbo].[StudentGrade] CHECK
CONSTRAINT [FK_StudentGrade_Course]
GO

--Define the relationship between StudentGrade and Student.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_StudentGrade_Student]'))
AND parent_object_id = OBJECT_ID(N'[dbo].[StudentGrade]'))
ALTER TABLE [dbo].[StudentGrade] WITH CHECK ADD
CONSTRAINT [FK_StudentGrade_Student] FOREIGN KEY([StudentID])
REFERENCES [dbo].[Person] ([PersonID])
GO
ALTER TABLE [dbo].[StudentGrade] CHECK
CONSTRAINT [FK_StudentGrade_Student]
GO

-- Define the relationship between CourseInstructor and Course.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_CourseInstructor_Course]'))
AND parent_object_id = OBJECT_ID(N'[dbo].[CourseInstructor]'))
ALTER TABLE [dbo].[CourseInstructor] WITH CHECK ADD
CONSTRAINT [FK_CourseInstructor_Course] FOREIGN KEY([CourseID])
REFERENCES [dbo].[Course] ([CourseID])
GO
ALTER TABLE [dbo].[CourseInstructor] CHECK
CONSTRAINT [FK_CourseInstructor_Course]
GO

-- Define the relationship between CourseInstructor and Person.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_CourseInstructor_Person]'))
AND parent_object_id = OBJECT_ID(N'[dbo].[CourseInstructor]'))
ALTER TABLE [dbo].[CourseInstructor] WITH CHECK ADD
CONSTRAINT [FK_CourseInstructor_Person] FOREIGN KEY([PersonID])
REFERENCES [dbo].[Person] ([PersonID])
GO
ALTER TABLE [dbo].[CourseInstructor] CHECK
CONSTRAINT [FK_CourseInstructor_Person]
GO

-- Define the relationship between Course and Department.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_Course_Department]'))
AND parent_object_id = OBJECT_ID(N'[dbo].[Course]'))
ALTER TABLE [dbo].[Course] WITH CHECK ADD
CONSTRAINT [FK_Course_Department] FOREIGN KEY([DepartmentID])
REFERENCES [dbo].[Department] ([DepartmentID])
GO
ALTER TABLE [dbo].[Course] CHECK CONSTRAINT [FK_Course_Department]
GO

--Define the relationship between OfficeAssignment and Person.
IF NOT EXISTS (SELECT * FROM sys.foreign_keys
WHERE object_id = OBJECT_ID(N'[dbo].[FK_OfficeAssignment_Person]'))
AND parent_object_id = OBJECT_ID(N'[dbo].[OfficeAssignment]'))
ALTER TABLE [dbo].[OfficeAssignment] WITH CHECK ADD
CONSTRAINT [FK_OfficeAssignment_Person] FOREIGN KEY([InstructorID])
REFERENCES [dbo].[Person] ([PersonID])
GO
ALTER TABLE [dbo].[OfficeAssignment] CHECK
CONSTRAINT [FK_OfficeAssignment_Person]
GO

-- Create InsertOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[InsertOfficeAssignment]'))

```

```

AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[InsertOfficeAssignment]
@InstructorID int,
@Location nvarchar(50)
AS
INSERT INTO dbo.OfficeAssignment (InstructorID, Location)
VALUES (@InstructorID, @Location);
IF @@ROWCOUNT > 0
BEGIN
SELECT [Timestamp] FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
END
'
END
GO

--Create the UpdateOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[UpdateOfficeAssignment]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[UpdateOfficeAssignment]
@InstructorID int,
@Location nvarchar(50),
@OrigTimestamp timestamp
AS
UPDATE OfficeAssignment SET Location=@Location
WHERE InstructorID=@InstructorID AND [Timestamp]=@OrigTimestamp;
IF @@ROWCOUNT > 0
BEGIN
SELECT [Timestamp] FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
END
'
END
GO

-- Create the DeleteOfficeAssignment stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[DeleteOfficeAssignment]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[DeleteOfficeAssignment]
@InstructorID int
AS
DELETE FROM OfficeAssignment
WHERE InstructorID=@InstructorID;
'
END
GO

-- Create the DeletePerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[DeletePerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[DeletePerson]
@PersonID int
AS
DELETE FROM Person WHERE PersonID = @PersonID;
'
END
GO

```

```

-- Create the UpdatePerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[UpdatePerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[UpdatePerson]
@PersonID int,
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime,
@EnrollmentDate datetime,
@Discriminator nvarchar(50)
AS
UPDATE Person SET LastName=@LastName,
FirstName=@FirstName,
HireDate=@HireDate,
EnrollmentDate=@EnrollmentDate,
Discriminator=@Discriminator
WHERE PersonID=@PersonID;
'

END
GO

-- Create the InsertPerson stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[InsertPerson]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[InsertPerson]
@LastName nvarchar(50),
@FirstName nvarchar(50),
@HireDate datetime,
@EnrollmentDate datetime,
@Discriminator nvarchar(50)
AS
INSERT INTO dbo.Person (LastName,
FirstName,
HireDate,
EnrollmentDate,
Discriminator)
VALUES (@LastName,
@FirstName,
@HireDate,
@EnrollmentDate,
@Discriminator);
SELECT SCOPE_IDENTITY() as NewPersonID;
'

END
GO

-- Create GetStudentGrades stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects
WHERE object_id = OBJECT_ID(N'[dbo].[GetStudentGrades]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[GetStudentGrades]
@StudentID int
AS
SELECT EnrollmentID, Grade, CourseID, StudentID FROM dbo.StudentGrade
WHERE StudentID = @StudentID
'

END
GO

-- Create GetDepartmentName stored procedure.
IF NOT EXISTS (SELECT * FROM sys.objects

```

```

WHERE object_id = OBJECT_ID(N'[dbo].[GetDepartmentName]')
AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'
CREATE PROCEDURE [dbo].[GetDepartmentName]
@ID int,
@Name nvarchar(50) OUTPUT
AS
SELECT @Name = Name FROM Department
WHERE DepartmentID = @ID
'

END
GO

-- Insert data into the Person table.
USE School
GO
SET IDENTITY_INSERT dbo.Person ON
GO
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (1, 'Abercrombie', 'Kim', '1995-03-11', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (2, 'Barzdukas', 'Gytis', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (3, 'Justice', 'Peggy', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (4, 'Fakhouri', 'Fadi', '2002-08-06', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (5, 'Harui', 'Roger', '1998-07-01', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (6, 'Li', 'Yan', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (7, 'Norman', 'Laura', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (8, 'Olivotto', 'Nino', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (9, 'Tang', 'Wayne', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (10, 'Alonso', 'Meredith', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (11, 'Lopez', 'Sophia', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (12, 'Browning', 'Meredith', null, '2000-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (13, 'Anand', 'Arturo', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (14, 'Walker', 'Alexandra', null, '2000-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (15, 'Powell', 'Carson', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (16, 'Jai', 'Damien', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (17, 'Carlson', 'Robyn', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (18, 'Zheng', 'Roger', '2004-02-12', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (19, 'Bryant', 'Carson', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (20, 'Suarez', 'Robyn', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (21, 'Holt', 'Roger', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (22, 'Alexander', 'Carson', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (23, 'Morgan', 'Isaiah', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (24, 'Martin', 'Randall', null, '2005-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (25, 'Kanoor', 'Candace', '2001-01-15', null, 'Instructor');

```

```
VALUES (25, 'Rogers', 'Cody', null, '2001-01-15', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (26, 'Rogers', 'Cody', null, '2002-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (27, 'Serrano', 'Stacy', '1999-06-01', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (28, 'White', 'Anthony', null, '2001-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (29, 'Griffin', 'Rachel', null, '2004-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (30, 'Shan', 'Alicia', null, '2003-09-01', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (31, 'Stewart', 'Jasmine', '1997-10-12', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (32, 'Xu', 'Kristen', '2001-07-23', null, 'Instructor');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (33, 'Gao', 'Erica', null, '2003-01-30', 'Student');
INSERT INTO dbo.Person (PersonID, LastName, FirstName, HireDate, EnrollmentDate, Discriminator)
VALUES (34, 'Van Houten', 'Roger', '2000-12-07', null, 'Instructor');
GO
SET IDENTITY_INSERT dbo.Person OFF
GO
```

```
-- Insert data into the Department table.
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (1, 'Engineering', 350000.00, '2007-09-01', 2);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (2, 'English', 120000.00, '2007-09-01', 6);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (4, 'Economics', 200000.00, '2007-09-01', 4);
INSERT INTO dbo.Department (DepartmentID, [Name], Budget, StartDate, Administrator)
VALUES (7, 'Mathematics', 250000.00, '2007-09-01', 3);
GO
```

```
-- Insert data into the Course table.
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1050, 'Chemistry', 4, 1);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1061, 'Physics', 4, 1);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (1045, 'Calculus', 4, 7);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2030, 'Poetry', 2, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2021, 'Composition', 3, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (2042, 'Literature', 4, 2);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4022, 'Microeconomics', 3, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4041, 'Macroeconomics', 3, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (4061, 'Quantitative', 2, 4);
INSERT INTO dbo.Course (CourseID, Title, Credits, DepartmentID)
VALUES (3141, 'Trigonometry', 4, 7);
GO
```

```
-- Insert data into the OnlineCourse table.
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (2030, 'http://www.fineartschool.net/Poetry');
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (2021, 'http://www.fineartschool.net/Composition');
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (4041, 'http://www.fineartschool.net/Macroeconomics');
INSERT INTO dbo.OnlineCourse (CourseID, URL)
VALUES (3141, 'http://www.fineartschool.net/Trigonometry');
```

```

--Insert data into OnsiteCourse table.
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1050, '123 Smith', 'MTWH', '11:30');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1061, '234 Smith', 'TWHF', '13:15');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (1045, '121 Smith', 'MWHF', '15:30');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (4061, '22 Williams', 'TH', '11:15');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (2042, '225 Adams', 'MTWH', '11:00');
INSERT INTO dbo.OnsiteCourse (CourseID, Location, Days, [Time])
VALUES (4022, '23 Williams', 'MWF', '9:00');

-- Insert data into the CourseInstructor table.
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1050, 1);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1061, 31);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (1045, 5);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2030, 4);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2021, 27);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (2042, 25);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4022, 18);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4041, 32);
INSERT INTO dbo.CourseInstructor(CourseID, PersonID)
VALUES (4061, 34);
GO

--Insert data into the OfficeAssignment table.
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (1, '17 Smith');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (4, '29 Adams');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (5, '37 Williams');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (18, '143 Smith');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (25, '57 Adams');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (27, '271 Williams');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (31, '131 Smith');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (32, '203 Williams');
INSERT INTO dbo.OfficeAssignment( InstructorID, Location)
VALUES (34, '213 Smith');

-- Insert data into the StudentGrade table.
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 2, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2030, 2, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 3, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2030, 3, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 6, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 6, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 7, 3.0);

```

```
VALUES (2021, 7, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 7, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2021, 8, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (2042, 8, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 9, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 10, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 11, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 12, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 12, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 14, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 13, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 13, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 14, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 15, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 16, 2);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 17, null);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 19, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 20, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 21, 2);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 22, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4041, 22, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4061, 22, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (4022, 23, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1045, 23, 1.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 24, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 25, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 26, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 26, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 27, 3);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1045, 28, 2.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 28, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 29, 4);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1050, 30, 3.5);
INSERT INTO dbo.StudentGrade (CourseID, StudentID, Grade)
VALUES (1061, 30, 4);
GO
```


Entity Framework Tools & 扩展

2020/3/11 •

IMPORTANT

扩展由多种源生成，不作为实体框架的一部分进行维护。考虑使用第三方扩展时，请务必评估质量、授权、兼容性、支持等因素，确保其满足要求。

多年来，实体框架是一种常用的 O/RM。下面是在其开发的免费和付费工具和扩展的一些示例：

- [EF Power Tools 社区版](#)
- [EF 探查器](#)
- [ORM 探查器](#)
- [LINQPad](#)
- [LLBLGen Pro](#)
- [Huagati DBML/EDMX 工具](#)
- [Entity Developer](#)