

# Table of Contents

## Entity Framework

### 比较 EF Core 和 EF6

#### 哪个版本适合你

#### 功能比较

#### 同一应用程序中的 EF6 和 EF Core

#### 从 EF6 移植到 EF Core

## Entity Framework Core

### EF Core 中的新增功能

#### 入门

#### 创建模型

#### 查询数据

#### 保存数据

#### 支持的 .NET 实现

#### 数据库提供程序

#### 管理数据库架构

#### 命令行参考

#### 工具和扩展

#### 杂项

#### □ API 参考

## Entity Framework 6

#### □ 文档

#### □ API 参考

# Entity Framework 文档

## Entity Framework

Entity Framework 是一种支持 .NET 开发人员使用 .NET 对象处理数据库的对象关系映射程序 (O/RM)。它不要求提供开发人员通常需要编写的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

.NET Framework

.NET Core

ASP.NET Core

UWP

更多...



## 数据库提供程序

[SQL Server](#)

[MySQL](#)

[PostgreSQL](#)

[SQLite](#)

[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。



## 开始操作

[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

2018/3/21 • 1 min to read • [Edit Online](#)

Entity Framework 有两个不同版本, 即 Entity Framework Core 和 Entity Framework 6。

## Entity Framework 6

Entity Framework 6 (EF6) 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。它作为 .NET Framework 3.5 SP1 和 Visual Studio 2008 SP1 的一部分, 于 2008 年初次发布。从 EF4.1 版本开始, 它已经作为 [EntityFramework NuGet 包](#) 发布 - 目前是 NuGet.org 上最受欢迎的包之一。

EF6 产品仍受支持, 并将在未来一段时间内继续提供 bug 修复和细微改进。

## Entity Framework Core

Entity Framework Core (EF Core) 是轻量化、可扩展和跨平台版的 Entity Framework。与 EF6 相比, EF Core 引入了众多改进和新功能。不过, EF Core 是一个新代码基, 不如 EF6 成熟。

EF Core 保留了 EF6 的开发人员体验, 并且大多数顶级 API 也保持不变, 因此已使用过 EF6 的开发人员会感觉 EF Core 非常熟悉。不过, EF Core 是构建在一套全新的核心组件之上的。这意味着 EF Core 不会自动继承 EF6 的所有功能。虽然其中一些功能会在将来版本中提供, 但其他一些不太常用的功能不会在 EF Core 中提供。

凭借可扩展、轻量化的新核心, 我们还可以将一些 EF6 中不提供的功能添加到 EF Core(例如 LINQ 查询中的备用键、批量更新和混合客户端/数据库评估)。

# EF Core 和 EF6: 哪个版本适合你

2018/4/23 • 1 min to read • [Edit Online](#)

以下信息将帮助你在 Entity Framework Core 和 Entity Framework 6 之间作出选择。

## 针对新应用程序的选择指南

如果想要利用 EF Core 的所有功能且应用程序不需要任何 EF Core 中尚未实现的功能, 请考虑对新应用程序使用 EF Core。

EF6 需要 .NET Framework 4.0 (或更高版本) 且仅在 Windows 上受支持 (即它不在 .NET Core 上运行, 在其他操作系统中不受支持), 但是, 只要这些约束是可接受的, 且应用程序不需要 EF6 不适用的 EF Core 的新功能, 这对新应用程序仍是一个可行的选择。

查看[功能比较](#)以查看 EF Core 是否适合你的应用程序。

## 针对现有 EF6 应用程序的选择指南

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。如果想迁移到 EF Core 以使用新功能, 请确保在开始之前了解其限制。查看[功能比较](#)以查看 EF Core 是否适合你的应用程序。

你应该将从 EF6 移至 EF Core 视作迁移而不是升级。有关详细信息, 请参阅[从 EF6 到 EF Core 的迁移](#)。

# EF Core 和 EF6 逐个功能比较

2018/3/6 • 3 min to read • [Edit Online](#)

下表比较了 EF Core 和 EF6 中可用的功能。该表仅在高级别进行比较, 并没有列出每个功能, 也没有提供相同功能之间可能的差异详细信息。

EF Core 列包含功能首次出现的产品版本号。

创建模型	EF 6	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(查询类型)		2.1
约定	是	1.0
自定义约定	是	1.0(部分)
数据注释	是	1.0
Fluent API	是	1.0
继承: 每个层次结构一张表 (TPH)	是	1.0
继承: 每个类型一张表 (TPT)	是	
继承: 每个具体类一张表 (TPC)	是	
阴影状态属性		1.0
备用键		1.0
多对多, 无联接实体	是	
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	
模型的图形可视化效果	是	
图形模型编辑器	是	

创建模型	EF 6	EF CORE
模型格式:代码	是	1.0
模型格式:EDMX (XML)	是	
从数据库创建模型:命令行	是	1.0
从数据库创建模型:VS 向导	是	
从数据库更新模型	部分	
全局查询筛选器		2.0
表拆分	是	2.0
实体拆分	是	
数据库标量函数映射	差	2.0
字段映射		1.1
查询数据	EF6	EF Core
LINQ 查询	是	1.0(进行中, 针对复杂查询)
可读内容生成的 SQL	差	1.0
混合客户端/服务器评估		1.0
GroupBy 转换	是	2.1
加载相关数据:预先加载	是	1.0
加载相关数据:预先加载派生类型		2.1
加载相关数据:延迟加载	是	2.1
加载相关数据:显式加载	是	1.1
原始 SQL 查询:实体类型	是	1.0
原始 SQL 查询:非实体类型(例如查询类型)	是	2.1
原始 SQL 查询:使用 LINQ 编写		1.0
显式编译的查询	差	2.0
基于文本的查询语言(例如实体 SQL)	1.0	

创建模型	EF 6	EF CORE
保存数据	EF6	EF Core
更改跟踪:快照	是	1.0
更改追踪:通知	是	1.0
更改跟踪:代理	是	
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	
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)
其他功能	EF6	EF Core
迁移	是	1.0
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
生命周期挂钩(事件、截取)	是	
简单的日志记录(例如 Database.Log)	是	
DbContext 池		2.0
数据库提供程序	EF6	EF Core
SQL Server	是	1.0
MySQL	是	1.0
postgresql	是	1.0
Oracle	是	1.0 <sup>(1)</sup>



创建模型	EF 6	EF CORE
SQLite	是	1.0
SQL Server Compact	是	1.0 <sup>(2)</sup>
DB2	是	1.0
Firebird	是	2.0
Jet (Microsoft Access)		2.0 <sup>(2)</sup>
内存中(用于测试)		1.0
平台	EF6	EF Core
.NET framework(控制台、WinForms、WPF、ASP.NET)	是	1.0
.NET Core(控制台、ASP.NET Core)		1.0
Mono 和 Xamarin		1.0(进行中)
UWP		1.0(进行中)

<sup>1</sup> 目前提供付费提供程序。适用于 Oracle 的免费官方提供程序目前正在开发中。<sup>2</sup> 此提供程序仅适用于 .NET Framework(而不是 .NET Core)。

# 在同一个应用程序中使用 EF Core 和 EF6

2018/1/23 • 1 min to read • [Edit Online](#)

通过安装两个 NuGet 包可在同一个 .NET Framework 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。使用命名空间别名指令可以很容易地删除多义性, 例如:

```
using Microsoft.EntityFrameworkCore;  
using EF6 = System.Data.Entity; // e.g. EF6.DbContext
```

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。

# 从 EF6 移植到 EF Core

2018/1/23 • 1 min to read • [Edit Online](#)

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。

# 之前从 EF6 到 EF 核心的迁移：验证应用程序的要求

2018/1/23 • 3 min to read • [Edit Online](#)

在开始迁移过程之前很重要，以验证 EF 核心满足你的应用程序的数据访问要求。

## 缺少的功能

请确保 EF 核心具有所需应用程序中使用的所有功能。请参阅[功能比较](#)有关如何在 EF 核心中设置的功能比较到 ef6 更高版本的详细比较。如果缺少任何所需的功能，请确保，你可以补偿缺少这些功能移植到 EF 核心之前。

## 行为更改

这是从 EF6 和 EF 核心之间的行为中的某些更改非详尽列表。请务必记住这几点你端口作为你的应用程序因为它们可能更改你的应用程序的行为，但将不显示为编译错误后交换到 EF 核心的方式。

### DbSet.Add/Attach 和 graph 行为

在从 EF6，调用 `DbSet.Add()` 对实体的所有实体在其导航属性中引用的递归搜索。发现了，并且不由上下文中，已跟踪的任何实体也是标记为添加。`DbSet.Attach()` 行为相同，但所有实体都标记为不变。

EF 核心执行类似的递归搜索，但使用一些略有不同的规则。

- 根实体始终处于请求的状态 (为添加 `DbSet.Add` 和不变 `DbSet.Attach`)。
- 适用于导航属性在递归搜索过程中找到的实体：
  - 如果实体的主键是生成的存储
    - 如果主键不设置为一个值，则状态将设置为添加。主键值被视为“未设置”被分配的属性类型的 CLR 默认值 (即 `0` 为 `int`，`null` 为 `string` 等。 )。
    - 如果为主键设置为一个值，状态将是设置为不变。
    - 如果主键不是数据库生成，则会将实体放在与根相同的状态。

### 第一个数据库初始化的代码

从 EF6 具有大量的幻它围绕选择数据库连接和初始化数据库执行。这些规则包括：

- 如果不执行任何配置，则从 EF6 将选择 SQL Express 或 LocalDb 上的数据库。
- 如果连接字符串具有相同名称作为上下文是在应用程序 `App/Web.config` 文件，将使用此连接。
- 如果数据库不存在，则创建它。
- 如果没有任何从模型表的数据库中存在，则会将当前模型的架构添加到数据库。如果启用迁移后，它们用于创建数据库。
- 如果数据库存在并从 EF6 以前已创建了架构，然后针对与当前模型的兼容性检查架构。如果自创建架构时，该模型已更改，将引发异常。

EF 核心不执行任何此幻数。

- 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()` 将迁移 (或 `DbContext.Database.EnsureCreated()` 和 `EnsureDeleted()` 而无需使用迁移的创建或删除数据库)。

## 代码的第一个表命名约定

从 EF6 通过计算该实体映射到的默认表名称的复数形式服务运行的实体类名。

EF 核心使用的名称的 `DbSet` 派生上下文公开了实体的属性。如果该实体没有 `DbSet` 使用属性, 则类名称。

# 移植到 EF 核心了从 EF6 基于 EDMX 的模型

2018/5/3 • 1 min to read • [Edit Online](#)

EF 核心不支持的模型的 EDMX 文件格式。若要移植这些模型中，最佳选择是从你的应用程序数据库中生成新的基于代码的模型。

## 安装 EF 核心 NuGet 包

安装 `Microsoft.EntityFrameworkCore.Tools` NuGet 包。

## 重新生成模型

反向工程功能现在可用于创建基于现有数据库的模型。

在 Package Manager Console 中运行以下命令 (工具 -> NuGet 包管理器 -> 程序包管理器控制台)。请参阅 [Package Manager Console \(Visual Studio\)](#) 有关命令选项搭建基架的表等等的子集。

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

例如，下面是要从 SQL Server LocalDB 实例上的博客数据库模型搭建基架命令。

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

## 删除从 EF6 模型

现在，你将删除从 EF6 模型从你的应用程序。

最好将从 EF6 NuGet 包 (EntityFramework) 安装，因为 EF 核心和 ef6 更高版本可以是在同一应用程序并行使用。但是，如果你不想要在你的应用程序的任何区域中使用 ef6 更高版本，然后卸载程序包将有助于为提供上的代码片段，需要注意的编译错误。

## 更新你的代码

此时，它是一个寻址编译错误和查看代码，以查看你会影响从 EF6 和 EF 核心之间的行为更改。

## 测试端口

只是因为你的应用程序将编译，并不意味着它成功移植到 EF 核心。你将需要测试应用程序，以确保任何行为更改产生负面影响你的应用程序的所有区域。

### 提示

请参阅 [Getting Started with ASP.NET 核心使用现有的数据库上的 EF 核心](#) 有关现有数据库，使用方式的其他参考

# 移植到 EF 核心 EF6 基于代码的模型

2018/1/23 • 2 min to read • [Edit Online](#)

如果你已阅读所有警告，并且就可以准备移植，以下是一些帮助你开始准则。

## 安装 EF 核心 NuGet 包

若要使用 EF 核心，请为你想要使用的数据库提供程序安装 NuGet 包。例如，如果目标 SQL Server，你将安装 `Microsoft.EntityFrameworkCore.SqlServer`。请参阅[的数据库提供程序](#)有关详细信息。

如果你打算使用迁移，则还应安装 `Microsoft.EntityFrameworkCore.Tools` 包。

最好将从 EF6 NuGet 包 (EntityFramework) 安装，因为 EF 核心和 ef6 更高版本可以是在同一应用程序并行使用。但是，如果你不想要在你的应用程序的任何区域中使用 ef6 更高版本，然后卸载程序包将有助于为提供上的代码片段，需要注意的编译错误。

## 交换命名空间

你使用 ef6 更高版本中的大多数 Api 位于 `System.Data.Entity` 命名空间（和相关子命名空间）。第一个代码更改是切换到 `Microsoft.EntityFrameworkCore` 命名空间。你将通常从你派生的上下文的代码文件开始，然后制定据此，在发生解决编译错误。

## 上下文配置（连接等。）

中所述[确保 EF 核心将工作为应用程序](#)，EF 核心具有较少幻解决检测要连接到的数据库。你将需要重写 `OnConfiguring` 派生的上下文和使用的数据库提供程序特定的 API 来设置连接到数据库上的方法。

大多数 ef6 更高版本应用程序将连接字符串存储在应用程序 `App/Web.config` 文件。在 EF 核，读取此连接字符串使用 `ConfigurationManager` API。你可能需要添加对的引用 `System.Configuration` framework 程序集要能够使用此 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);
    }
}
```

## 更新你的代码

此时，它是一种解决编译错误，以及评审代码，以查看是否行为更改将影响你。

## 现有的迁移

实际上，没有端口现有 ef6 更高版本迁移到 EF 核心的可行方法。

如果可能，最好假定从 ef6 更高版本的所有以前迁移已应用到数据库，然后从，迁移架构的开始点使用 EF 核心。若要执行此操作，你将使用 `Add-Migration` 命令添加迁移后模型移植到 EF 核心。然后，你将删除的所有代码 `Up` 和

Down 方法的基架的迁移。该初始迁移已基架时，后续迁移将对模型进行比较。

## 测试端口

只是因为你的应用程序将编译，并不意味着它成功移植到 EF 核心。你将需要测试应用程序，以确保任何行为更改产生负面影响你的应用程序的所有区域。



# Entity Framework Core 快速概览

2018/6/5 • 2 min to read • [Edit Online](#)

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

如果希望通过编写代码进行了解，建议你阅读其中一篇[入门](#)指南以开始使用 EF Core。

## EF Core 中的新增功能

如果已熟悉 EF Core 并希望直接了解新版本的详细信息，请参阅：

- [EF Core 2.1中的新增功能](#)
- [将现有应用程序升级至 EF Core 2.x](#)

## 获取 Entity Framework Core

针对要使用的数据库提供程序，[安装 NuGet 包](#)。例如，要在跨平台开发中安装 SQL Server 提供程序，请在命令中使用 `dotnet` 工具：

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

或者在 Visual Studio 中，使用包管理器控制台：

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

有关可用提供程序的信息，请参阅[数据库提供程序](#)；有关更多详细的安装步骤，请参阅[安装 EF Core](#)。

## 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派生上下文构成，用于查询和保存数据。有关详细信息，请参阅[创建模型](#)。

可从现有数据库生成模型，手动编码模型使之与数据库相匹配，或使用 EF 迁移基于模型创建数据库（并在模型随时间推移发生更改后进行相应改进）。

```

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=MyDatabase;Trusted_Connection=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();
}

```

# EF Core 中的新增功能

2018/6/11 • 1 min to read • [Edit Online](#)

可使用以下链接了解每个版本中的新功能：

## 未来将要发布的版本

- [EF Core 路线图](#)

## 最新版本

- [EF Core 2.1 \(最新发布版本\)](#)
- [EF Core 2.0](#)

## 以前的版本

- [EF Core 1.1](#)
- [EF Core 1.0](#)

# Entity Framework Core 路线图

2018/4/25 • 4 min to read • [Edit Online](#)

## 重要事项

请注意，将来版本的功能集和计划会发生更改，尽管我们会尽力使此页面保持最新状态，但它可能不会始终反映我们的最新计划。

EF Core 2.1 的第二个预览版发布于 2018 年 4 月。现在可在 [EF Core 2.1 中的新增功能](#) 中查找有关此版本的详细信息。

我们计划每月发布 EF Core 2.1 的其他预览版，并于 2018 年的第二个日历季度发布最终版。

我们尚未完成 2.1 后的下一个版本的[发布规划](#)。

## 计划

EF Core 的计划与 [.NET Core 计划](#) 以及 [ASP.NET Core 计划](#) 同步。

## 积压工作 (backlog)

我们在问题跟踪程序中使用[积压工作 \(backlog\) 里程碑](#)，维护问题和功能的详细列表。客户可评论并投票。

如果我们合理地预期我们将在某个时候处理某个问题，或者社区成员可解决该问题，我们倾向于保持问题的待解决状态，但这并不意味着我们计划在特定时间范围内解决该问题，除非我们在[版本规划过程](#)中为其分配了特定的里程碑。

如果我们没有计划实现某项功能，可能会关闭该问题。对于已关闭的问题，如果我们稍后获得相关新信息，可重新考虑。

所有这一切表明，我们没有足够的信息可以预测未来，我们不会说我们在 Y 时间/版本中将提供功能 X。正如在所有软件项目中，可以随时更改优先级、发布计划和可用资源等。

## 版本规划过程

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。积压工作 (backlog) 肯定不会自动转换成版本计划。EF6 中功能的状态也并不意味着需要在 EF Core 中实现该功能。

此处很难详细说明版本计划所遵循的整个过程，这一方面是因为主要涉及到特定功能、机会和优先级，另一方面是因为过程会随着每个版本更改。但是，在确定下一步工作内容时，总结要解决的常见问题相对容易：

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？我们汇总了众多来源(其中包括对问题的评论和投票)的反馈。
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，许多开发人员可以通过映射联接表格，暂时避开缺少多对多本机支持的问题。显而易见，并非所有开发人员都能这样做，但这是一个重要方法。
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？我们往往喜欢可充当其他功能的构建基块的功能，例如，用于自有类型的表拆分有助于实现 TPT 支持。
4. 功能是可扩展性点吗？我们往往喜欢可扩展性点，因为它们能让开发人员更轻松地将自己的行为，并通过该方式获得一些缺少的功能。我们计划添加部分功能，作为延迟加载工作的开始。

5. **该功能与其他产品结合使用时的增效作用如何？** 我们往往喜欢允许 EF 与其他产品结合使用或可显著改善使用其他产品体验的功能，如 .NET Core、最新版 Visual Studio、Microsoft Azure 等。
6. **从事功能开发工作的人员的能力如何？如何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拥有不同领域的不同程度的经验，我们需要进行相应地计划。即便我们希望“全员就位”开发特定功能，如 GroupBy 转换或多对多功能，但这是不切实际的。

正如前文所述，这个过程会随着每个版本变化，将来我们希望为开发人员社区的成员提供更多的机会，为版本计划出谋划策，例如，使查看功能和版本计划的建议草稿更加轻松。

# EF Core 2.1 中的新增功能

2018/4/25 • 7 min to read • [Edit Online](#)

## 注意

此版本仍处于预览版。

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 现已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 运算符始终是在内存中进行计算。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 => 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。

## OwnedAttribute

现只需使用 `[Owned]` 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]
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; }
}
```

## 全新 dotnet-ef 全局工具

dotnet-ef 命令已改为 .NET CLI 全局工具, 因此无须在项目中使用 DotNetCliToolReference, 也可以使用各项迁移, 或通过现有数据库搭建 DbContext 基架。

## 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, 预览 1 中



引入的 [Owned] 属性已移到此包中。

## 状态更改事件

`ChangeTracker` 中新增的 `Tracked` 和 `StateChanged` 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

EF Core 2.1 旨在与针对 EF Core 2.0 创建的数据库提供程序兼容，或至少要求改动最少。虽然需要最新提供程序方可使用上述某些功能(如值转换)，但现有提供程序可使用其他一些功能(例如延迟加载)。

### 提示

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

2018/3/6 • 11 min to read • [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");
```

### 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 CLR 类型, 但是由于它不能仅由 CLR 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`。模型级别筛选器将使用来自正确上下文实例的值。即执行查询的值。

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 Exception();
    }
}
```

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;
```

需要注意以下事项：

- 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, 原因是 EF Core 迁移不负责创建映射函数

### 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<BloggingContext>(
    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是否有可用的实例。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。

#### 警告

如果需要在派生 DbContext 类(不应该在请求之间共享)中保持自己的状态(例如私有字段), 请避免使用 DbContext 池。EF Core 只会重置将 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, 同时会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, 采用这种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。

下面是一个示例:

```
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 查询中调用它们。此类方法的第一个示例是 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 `.Relational`（而不是 `.SqlServer`、`.Sqlite` 等）来访问常见关系元数据概念。

### 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

2018/1/23 • 1 min to read • [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。

## Change tracking

### 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

2018/1/23 • 4 min to read • [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

可以添加到实体类/属性并可影响 EF 模型的属性 (即添加 [必需] 将让 EF 知道属性是必需的)。

### 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。

### UNIQUE 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的错误消息。

# Change tracking

## 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。

## 通知更改追踪

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。

## 开放式并发

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

当上下文不需要监视实体实例的变化(即结果是只读的)时，可加快查询执行速度。

## 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](#)。

# Entity Framework Core 入门

2018/5/17 • 1 min to read • [Edit Online](#)

## 安装 EF Core

将 EF Core 添加到不同平台和常用 IDE 中的应用程序的所需步骤汇总。

## 分步入门教程

无需具备 Entity Framework Core 或特定 IDE 背景知识即可学习这些入门教程。这些教程会逐步演示如何创建简单的应用程序，以便查询和保存数据库中的数据。我们已提供许多教程，指导你开始使用各种操作系统和应用程序类型。

Entity Framework Core 可基于现有数据库创建模型，也可基于模型创建数据库。提供的教程演示了这两种方法。

### 注意

这些教程和随附示例已经更新为使用 EF Core 2.0 (UWP 教程除外，此教程仍然使用 EF Core 1.1)。但在大多数情况下，应该可以使用以前的版本创建应用程序，只需对指令进行极少修改。

# 安装 EF Core

2018/5/23 • 6 min to read • [Edit Online](#)

## 先决条件

为了开发 .NET Core 2.0 应用程序(包括面向 .NET Core 的 ASP.NET Core 2.0 应用程序), 需要下载并安装适合所用平台的 [.NET Core 2.0 SDK](#) 版本。即使已安装 Visual Studio 2017 版本 15.3, 也是如此。

为了将 EF Core 2.0 或任何其他 .NET Standard 2.0 库用于非 .NET Core 2.0 版本的 .NET 平台(例如 .NET Framework 4.6.1 或更高版本), 需要可识别 .NET Standard 2.0 及其兼容框架的 NuGet 版本。可通过几种方法获取此版本:

- 安装 Visual Studio 2017 版本 15.3
- 如果使用 Visual Studio 2015, 请[下载 NuGet 客户端并升级至 3.6.0 版](#)

为了与 .NET Framework 2.0 库兼容, 使用以前版本的 Visual Studio 创建且面向 .NET Framework 的项目可能需要其他修改:

- 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, 还要确保存在以下条目:

```
<GenerateBindingRedirectsOutputType>true</GenerateBindingRedirectsOutputType>
```

## 获取相应工具

将 EF Core 运行时库添加到应用程序的推荐方法是, 从 NuGet 安装 EF Core 数据库提供程序。

除运行时库外, 还可安装相应工具, 以便在设计时更轻松地执行项目中的多个 EF Core 相关任务, 例如创建和应用迁移以及基于现有数据库创建模型。

### 提示

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。

### 提示

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。面向 ASP.NET Core 早期版本的应用程序需升级至 ASP.NET Core 2.0 方可使用 EF Core 2.0。

## 使用 .NET Core 命令行接口 (CLI) 进行跨平台开发

要开发面向 [.NET Core](#) 的应用程序, 可以选择将 [dotnet](#) CLI 命令与你喜爱的文本编辑器或集成开发环境 (IDE) 结合使用, 例如 Visual Studio、Visual Studio for Mac 或 Visual Studio Code。

#### 重要事项

面向 .NET Core 的应用程序需要使用特定版本的 Visual Studio，例如 .NET Core 1.x 开发需要使用 Visual Studio 2017，而 .NET Core 2.0 开发需要使用 Visual Studio 2017 版本 15.3。

要在跨平台 .NET Core 应用程序中安装或升级 SQL Server 提供程序，请切换至应用程序的目录，并在命令行中运行以下命令：

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

可以使用 `-v` 修饰符在 `dotnet add package` 命令中指明特定的安装版本。例如 要安装 EF Core 2.0 包，请将 `-v 2.0.0` 追加到此命令。

EF Core 包含一套以 `dotnet ef` 开头的附加 `dotnet` CLI 命令。要使用 `dotnet ef` CLI 命令，应用程序的 `.csproj` 文件需包含以下条目：

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

用于 EF Core 的 .NET Core CLI 工具还需要一个名为 `Microsoft.EntityFrameworkCore.Design` 的单独包。可使用以下命令将其轻松添加至项目：

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

#### 重要事项

请务必使用与运行时包主版本相匹配的工具包版本。

## Visual Studio 开发

使用 Visual Studio 可以开发许多不同类型的应用程序，这些应用程序面向 .NET Core、.NET Framework 或受 EF Core 支持的其他平台。

使用 Visual Studio 可通过两种方式在应用程序中安装 EF Core 数据库提供程序：

#### 使用 NuGet 的包管理器用户界面

- 在菜单上选择“项目”>“管理 NuGet 程序包”
- 单击“浏览”或“更新”选项卡
- 选择 `Microsoft.EntityFrameworkCore.SqlServer` 包及所需版本，然后确认

#### 使用 NuGet 的包管理器控制台 (PMC)

- 在菜单上选择“工具”>“NuGet 包管理器”>“包管理器控制台”
- 在 PMC 中键入并运行以下命令：

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

- 可以改为使用 `Update-Package` 命令将已安装的包更新至较新版本
- 要指定特定版本，可使用 `-Version` 修饰符；例如，要安装 EF Core 2.0 包，可将 `-Version 2.0.0` 追加到该命令

## 工具

Visual Studio 中还有 PowerShell 版本的在 [PMC 内运行的 EF Core 命令](#)，其功能与 `dotnet ef` 命令相似。要使用这些命令，请使用包管理器 UI 或 PMC 安装 `Microsoft.EntityFrameworkCore.Tools` 包。

### 重要事项

请务必使用与运行时包主版本相匹配的工具包版本。

### 提示

尽管可以在 Visual Studio 中使用来自 PMC 的 `dotnet ef` 命令，但使用 PowerShell 版本要方便得多：

- 它们会自动使用 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。

### 重要事项

**EF Core 2.0 中的弃用包：**如果将现有应用程序升级至 EF Core 2.0，可能需要手动删除一些对旧版 EF Core 包的引用。具体而言，EF Core 2.0 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。

# 在 .NET Framework 上开始使用 EF Core

2018/2/13 • 1 min to read • [Edit Online](#)

无需具备 Entity Framework Core 或 Visual Studio 背景知识即可学习这 101 个教程。这些教程会逐步演示如何创建简单的 .NET Framework 控制台应用程序，以便查询和保存数据库中的数据。你可以选择一个教程，了解如何基于现有数据库创建模型，或者基于模型创建数据库。

可将在这些教程中学到的技术应用于任何面向 .NET Framework (包括 WPF 和 WinForms) 的应用程序。

## 注意

这些教程和随附示例已更新为使用 EF Core 2.0。但在大多数情况下，应该可以使用以前的版本创建应用程序，只需对指令进行极少修改。



# 使用新数据库在 .NET Framework 上开始使用 EF Core

2018/5/15 • 3 min to read • [Edit Online](#)

在本演练中，你将构建一个控制台应用程序，该程序使用 Entity Framework 对 Microsoft SQL Server 数据库执行基本数据访问。你将使用迁移功能从模型创建数据库。

## 提示

可在 [GitHub](#) 上查看此文章的示例。

## 系统必备

完成本演练需要以下先决条件：

- [Visual Studio 2017](#)
- [最新版本的 NuGet 包管理器](#)
- [最新版本的 Windows PowerShell](#)

## 创建新项目

- 打开 Visual Studio
- “文件”>“新建”>“项目...”
- 从左侧菜单中选择“模板”>“Visual C#”>“Windows 经典桌面”
- 选择“控制台应用(.NET Framework)”项目模板
- 确保面向 .NET Framework 4.5.1 或更高版本
- 为项目提供名称，然后单击“确定”

## 安装 Entity Framework

要使用 EF Core，请为要作为目标对象的数据库提供程序安装程序包。本演练使用 SQL Server。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- “工具”>“NuGet 包管理器”>“包管理器控制台”
- 运行 `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

在本演练的后面部分，我们还将一些 Entity Framework Tools 用于维护数据库。因此，我们也会安装此工具包。

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

## 创建你自己的模型

现在是时候定义构成模型的上下文和实体类了。

- “项目”>“添加类...”

- 输入“Model.cs”作为名称, 然后单击“确定”
- 将此文件的内容替换为以下代码

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

namespace EFGetStarted.ConsoleApp
{
    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=EFGetStarted.ConsoleApp.NewDb;Trusted_Connection=True;");
        }
    }

    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; }
    }
}
```

#### 提示

在实际的应用程序中, 你可以将每个类放在单独的文件中, 并将连接字符串放在 `App.Config` 文件中, 然后使用 `ConfigurationManager` 将其读出。为简单起见, 在本教程中, 我们将所有内容都放在单个代码文件中。

## 创建数据库

现在你已经有了模型, 可以使用迁移为自己创建数据库。

- “工具”->“NuGet 包管理器”->“包管理器控制台”
- 运行 `Add-Migration MyFirstMigration` 来为迁移搭建基架, 从而为模型创建一组初始表。
- 运行 `Update-Database` 以将新迁移应用到数据库。由于数据库尚不存在, 因此将在应用迁移之前进行创建。

### 提示

如果将来对模型进行更改, 则可以使用 `Add-Migration` 命令为新迁移搭建基架, 以便对数据库进行相应的架构更改。检查已搭建基架的代码(并进行了必要更改)后, 就可以使用 `Update-Database` 命令将更改应用到数据库。

EF 使用数据库中的 `__EFMigrationsHistory` 表来跟踪哪些迁移已经应用到数据库。

## 使用模型

你现在可以使用模型执行数据访问。

- 打开 Program.cs
- 将此文件的内容替换为以下代码

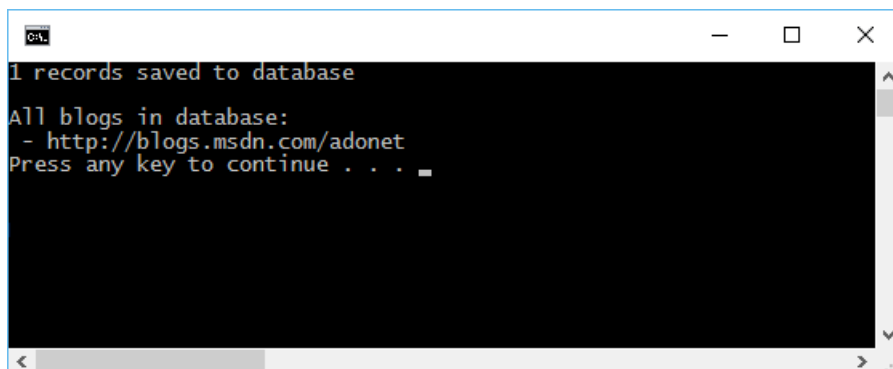
```
using System;

namespace EFGetStarted.ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BloggingContext())
            {
                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}
```

- “调试”>“开始执行(不调试)”

你将看到, 把一个博客保存到数据库后, 所有博客的详细信息都将显示在控制台中。



# 通过现有数据库在 .NET Framework 上开始使用 EF Core

2018/5/15 • 4 min to read • [Edit Online](#)

在本演练中，你将构建一个控制台应用程序，该程序使用 Entity Framework 对 Microsoft SQL Server 数据库执行基本数据访问。你将使用反向工程基于现有数据库创建 Entity Framework 模型。

## 提示

可在 [GitHub](#) 上查看此文章的示例。

## 系统必备

完成本演练需要以下先决条件：

- [Visual Studio 2017](#)
- [最新版本的 NuGet 包管理器](#)
- [最新版本的 Windows PowerShell](#)
- [博客数据库](#)

### 博客数据库

本教程使用 LocalDb 实例上的博客数据库作为现有数据库。

## 提示

如果你已在其他教程中创建了博客数据库，则可以跳过这些步骤。

- 打开 Visual Studio
- “工具”->“连接到数据库...”
- 选择“Microsoft SQL Server”，然后单击“继续”
- 输入“(localdb)\mssqllocaldb”作为服务器名称
- 输入“master”作为数据库名称，然后单击“确定”
- Master 数据库现在显示在“服务器资源管理器”的“数据连接”中
- 右键单击“服务器资源管理器”中的数据库，然后选择“新建查询”
- 将下面列出的脚本复制到查询编辑器中
- 右键单击查询编辑器，然后选择“执行”

```

CREATE DATABASE [Blogging];
GO

USE [Blogging];
GO

CREATE TABLE [Blog] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Post] (
    [PostId] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    [Content] nvarchar(max),
    [Title] nvarchar(max),
    CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog] ([BlogId]) ON DELETE CASCADE
);
GO

INSERT INTO [Blog] (Url) VALUES
('http://blogs.msdn.com/dotnet'),
('http://blogs.msdn.com/webdev'),
('http://blogs.msdn.com/visualstudio')
GO

```

## 创建新项目

- 打开 Visual Studio
- “文件”>“新建”>“项目...”
- 从左侧菜单中选择“模板”>“Visual C#”>“Windows”
- 选择“控制台应用程序”项目模板
- 确保面向 .NET Framework 4.5.1 或更高版本
- 为项目提供名称，然后单击“确定”

## 安装 Entity Framework

要使用 EF Core，请为要作为目标对象的数据库提供程序安装程序包。本演练使用 SQL Server。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- “工具”>“NuGet 包管理器”>“包管理器控制台”
- 运行 `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

若要从现有数据库启用反向工程，我们还需要安装几个其他程序包。

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

## 对模型实施反向工程

现在是时候基于现有数据库创建 EF 模型了。

- “工具”->“NuGet 包管理器”->“包管理器控制台”

- 运行以下命令以从现有数据库创建模型

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

反向工程过程基于现有数据库的架构创建实体类和派生上下文。实体类是简单的 C# 对象, 代表要查询和保存的数据。

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

namespace EFGetStarted.ConsoleApp.ExistingDb
{
    public partial class Blog
    {
        public Blog()
        {
            Post = new HashSet<Post>();
        }

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

        public virtual ICollection<Post> Post { get; set; }
    }
}
```

上下文表示与数据库的会话, 并允许查询和保存实体类的实例。

```

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata;

namespace EFGetStarted.ConsoleApp.ExistingDb
{
    public partial class BloggingContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            #warning To protect potentially sensitive information in your connection string, you should move it
            out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection
            strings.
            optionsBuilder.UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;");
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>(entity =>
            {
                entity.Property(e => e.Url).IsRequired();
            });

            modelBuilder.Entity<Post>(entity =>
            {
                entity.HasOne(d => d.Blog)
                    .WithMany(p => p.Post)
                    .HasForeignKey(d => d.BlogId);
            });
        }

        public virtual DbSet<Blog> Blog { get; set; }
        public virtual DbSet<Post> Post { get; set; }
    }
}

```

## 使用模型

你现在可以使用模型执行数据访问。

- 打开 Program.cs
- 将此文件的内容替换为以下代码

```

using System;

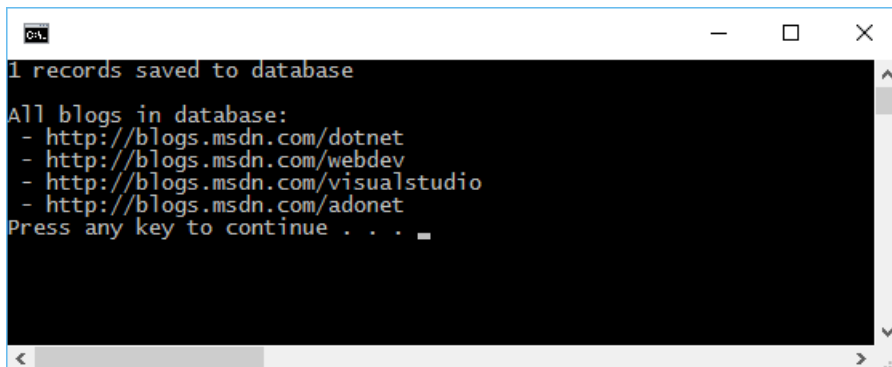
namespace EFGetStarted.ConsoleApp.ExistingDb
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BloggingContext())
            {
                db.Blog.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blog)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}

```

- “调试”>“开始执行(不调试)”

你将看到，把一个博客保存到数据库后，所有博客的详细信息都将显示在控制台中。



```

1 records saved to database

All blogs in database:
- http://blogs.msdn.com/dotnet
- http://blogs.msdn.com/webdev
- http://blogs.msdn.com/visualstudio
- http://blogs.msdn.com/adonet
Press any key to continue . . .

```



# 基于 .NET Core 的 EF Core 入门

2018/5/17 • 1 min to read • [Edit Online](#)

无需具备 Entity Framework Core 或 Visual Studio 背景知识即可学习这 101 个教程。这些教程会逐步演示如何创建简单的 .NET Core 控制台应用程序，以便查询和保存数据库中的数据。可在 .NET Core 支持的任何平台（Windows、OSX、Linux 等）上完成这些教程。

可在 [docs.microsoft.com/dotnet/articles/core](https://docs.microsoft.com/dotnet/articles/core) 找到 .NET Core 文档。

## 注意

这些教程和随附示例已经更新为使用 EF Core 2.0（UWP 教程除外，此教程仍然使用 EF Core 1.1）。但在大多数情况下，应该可以使用以前的版本创建应用程序，只需对指令进行极少修改。

# 使用新数据库在 .NET Core 控制台应用程序上开始使用 EF Core

2018/5/15 • 4 min to read • [Edit Online](#)

在本演练中，你将创建一个 .NET Core 控制台应用程序，该程序使用 Entity Framework Core 对 SQLite 数据库执行基本数据访问。你将使用迁移功能从模型创建数据库。有关使用 ASP.NET Core MVC 的 Visual Studio 版本，请参阅 [ASP.NET Core - 新数据库](#)。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 系统必备

完成本演练需要以下先决条件：

- 支持 .NET Core 的操作系统。
- [.NET Core SDK 2.0](#) (虽然可以使用指令并经过少量修改来创建较早版本的应用程序)。

## 创建新项目

- 为项目创建一个新的 `ConsoleApp.SQLite` 文件夹，并使用 `dotnet` 命令通过 .NET Core 应用程序对其进行填充。

```
mkdir ConsoleApp.SQLite
cd ConsoleApp.SQLite/
dotnet new console
```

## 安装 Entity Framework Core

要使用 EF Core，请为要作为目标对象的数据库提供程序安装程序包。本演练使用 SQLite。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- 安装 `Microsoft.EntityFrameworkCore.Sqlite` 和 `Microsoft.EntityFrameworkCore.Design`

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

- 手动编辑 `ConsoleApp.SQLite.csproj` 以将 `DotNetCliToolReference` 添加到 `Microsoft.EntityFrameworkCore.Tools.DotNet`：

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

`ConsoleApp.SQLite.csproj` 现在应包含以下代码：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="2.0.0" />
  </ItemGroup>
</Project>
```

注意:上面使用的版本号在发布时是正确的。

- 运行 `dotnet restore` 来安装新的程序包。

## 创建模型

定义构成模型的上下文和实体类:

- 使用以下内容创建一个新的 Model.cs 文件。

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

namespace ConsoleApp.SQLite
{
    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=logging.db");
        }
    }

    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; }
    }
}
```

提示:在实际的应用程序中,你可以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中。为简化本教程,我们会将所有内容放在一个文件中。

## 创建数据库

有了模型后，你可以使用[迁移](#)创建数据库。

- 运行 `dotnet ef migrations add InitialCreate` 以为迁移搭建基架，并为模型创建一组初始表。
- 运行 `dotnet ef database update` 以将新迁移应用到数据库。在应用迁移之前，此命令可创建数据库。

#### 注意

在 SQLite 中使用相对路径时，路径将与应用程序的主程序集相关。在此示例中，主要的二进制文件是

`bin/Debug/netcoreapp2.0/ConsoleApp.SQLite.dll`，因此 SQLite 数据库将在 `bin/Debug/netcoreapp2.0/blogging.db` 中。

## 使用模型

- 打开 Program.cs 并将内容替换为以下代码：

```
using System;

namespace ConsoleApp.SQLite
{
    public class Program
    {
        public static void Main()
        {
            using (var db = new BloggingContext())
            {
                db.Blogs.Add(new Blog { Url = "http://blogs.msdn.com/adonet" });
                var count = db.SaveChanges();
                Console.WriteLine("{0} records saved to database", count);

                Console.WriteLine();
                Console.WriteLine("All blogs in database:");
                foreach (var blog in db.Blogs)
                {
                    Console.WriteLine(" - {0}", blog.Url);
                }
            }
        }
    }
}
```

- 测试应用：

```
dotnet run
```

将一个博客保存到数据库，那么所有博客的详细信息都会显示在控制台中。

```
ConsoleApp.SQLite>dotnet run
1 records saved to database

All blogs in database:
- http://blogs.msdn.com/adonet
```

#### 更改模型：

- 如果对模型进行更改，则可以使用 `dotnet ef migrations add` 命令为新[迁移](#)搭建基架，以便对数据库进行相应的架构更改。检查已搭建基架的代码(并进行了必要更改)后，就可以使用 `dotnet ef database update` 命令将更改应用到数据库。
- EF 使用数据库中的 `__EFMigrationsHistory` 表来跟踪哪些迁移已经应用到数据库。
- 由于 SQLite 的限制，SQLite 不支持所有迁移(架构更改)。请参阅 [SQLite 限制](#)。进行新的开发时，考虑删除数

数据库并创建新的数据库, 而不是在模型更改时使用迁移。

## 其他资源

- - [通过 SQLite 新建数据库](#) - 跨平台的控制台 EF 教程。
- [Mac 或 Linux 上的 ASP.NET Core MVC 简介](#)
- [具有 Visual Studio 的 ASP.NET Core MVC 简介](#)
- [借助 Visual Studio 使用 ASP.NET Core 和 Entity Framework Core 入门](#)

# 基于 ASP.NET Core 的 EF Core 入门

2018/5/17 • 1 min to read • [Edit Online](#)

无需具备 Entity Framework Core 或 Visual Studio 背景知识即可学习这 101 个教程。这些教程会逐步演示如何创建简单的 ASP.NET Core 应用程序，以便查询和保存数据库中的数据。你可以选择一个教程，了解如何基于现有数据库创建模型，或者基于模型创建数据库。

可在 [docs.asp.net](https://docs.asp.net) 中找到 ASP.NET Core 文档。

## 注意

这些教程和随附示例已经更新为使用 EF Core 2.0 (UWP 教程除外，此教程仍然使用 EF Core 1.1)。但在大多数情况下，应该可以使用以前的版本创建应用程序，只需对指令进行极少修改。

# 使用新数据库在 ASP.NET Core 上开始使用 EF Core

2018/5/15 • 5 min to read • [Edit Online](#)

在本演练中，你将使用 Entity Framework Core 构建执行基本数据访问的 ASP.NET Core MVC 应用程序。你将使用迁移功能从 EF Core 模型创建数据库。有关更多 Entity Framework Core 教程的信息，请参阅[其他资源](#)。

本教程要求：

- 具有以下工作负载的 [Visual Studio 2017 15.3](#)：
  - “ASP.NET 和 Web 开发”（位于“Web 和云”下）
  - “.NET Core 跨平台开发”（位于“其他工具集”下）
- [.NET Core 2.0 SDK](#)。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 在 Visual Studio 2017 中创建新项目

- “文件”>“新建”>“项目”
- 从左侧菜单中选择“已安装”>“模板”>“Visual C#”>“.NET Core”。
- 选择“ASP.NET Core Web 应用程序”。
- 输入“EFGetStarted.AspNetCore.NewDb”作为名称，然后单击“确定”。
- 在“新建 ASP.NET Core Web 应用程序”对话框中：
  - 确保在下拉列表中选择“.NET Core”和“ASP.NET Core 2.0”选项
  - 选择“Web 应用程序(模型视图控制器)”项目模板
  - 确保将“身份验证”设置为“无身份验证”
  - 单击“确定”

警告：如果你使用“个人用户帐户”而不是“无身份验证”，则会将 Entity Framework Core 模型添加到

`Models\IdentityModel.cs` 中的项目中。使用本演练中学习的技巧，你可以选择添加第二个模型，或者扩展此现有模型以包含实体类。

## 安装 Entity Framework Core

为要作为目标对象的 EF Core 数据库提供程序安装程序包。本演练使用 SQL Server。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- “工具”>“NuGet 包管理器”>“包管理器控制台”
- 运行 `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

我们将使用一些 Entity Framework Core 工具从 EF Core 模型创建数据库。因此，我们也会安装此工具包：

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

我们稍后将使用一些 ASP.NET Core 基架工具来创建控制器和视图。因此，我们也会安装此设计包：

- 运行 `Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design`

## 创建模型

定义构成模型的上下文和实体类：

- 右键单击“Models”文件夹，然后选择“添加”>“类”。
- 输入“Model.cs”作为名称，然后单击“确定”。
- 将此文件的内容替换为以下代码：

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

namespace EFGetStarted.AspNetCore.NewDb.Models
{
    public class BloggingContext : DbContext
    {
        public BloggingContext(DbContextOptions<BloggingContext> options)
            : base(options)
        { }

        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; }

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

注意：在真正的应用程序中，通常会将模型中的每个类放在单独的文件中。为简单起见，我们在本教程中将所有类放在一个文件中。

## 通过依赖关系注入注册上下文

服务(例如 `BloggingContext`)在应用程序启动期间通过[依赖关系注入](#)进行注册。然后，通过构造函数参数或属性为需要这些服务的组件(如 MVC 控制器)提供相应服务。

为了使 MVC 控制器能够使用 `BloggingContext`，我们将把它注册为服务。

- 打开 Startup.cs
- 添加以下 `using` 语句：

```
using EFGetStarted.AspNetCore.NewDb.Models;
using Microsoft.EntityFrameworkCore;
```



添加 `AddDbContext` 方法将其注册为服务：

- 将以下代码添加到 `ConfigureServices` 方法：

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    var connection = @"Server=
(localdb)\mssqllocaldb;Database=EFGetStarted.AspNetCore.NewDb;Trusted_Connection=True;ConnectRetryCount=
0";
    services.AddDbContext<BlogggingContext>(options => options.UseSqlServer(connection));
}
```

注意：真正的应用程序通常会连接字符串放在配置文件中。为了简单起见，我们在代码中对它进行定义。有关详细信息，请参阅[连接字符串](#)。

## 创建数据库

有了模型后，你可以使用[迁移](#)创建数据库。

- 打开 PMC：

“工具”->“NuGet 包管理器”->“包管理器控制台”

- 运行 `Add-Migration InitialCreate` 来为迁移搭建基架，从而为模型创建一组初始表。如果收到错误，指出 `The term 'add-migration' is not recognized as the name of a cmdlet`，请关闭并重新打开 Visual Studio。
- 运行 `Update-Database` 以将新迁移应用到数据库。在应用迁移之前，此命令可创建数据库。

## 创建控制器

在项目中启用基架：

- 在“解决方案资源管理器”中，右键单击“控制器”文件夹，然后选择“添加”>“控制器”。
- 选择“最小依赖项”，然后单击“添加”。
- 你可以忽略或删除 `ScaffoldingReadMe.txt` 文件。

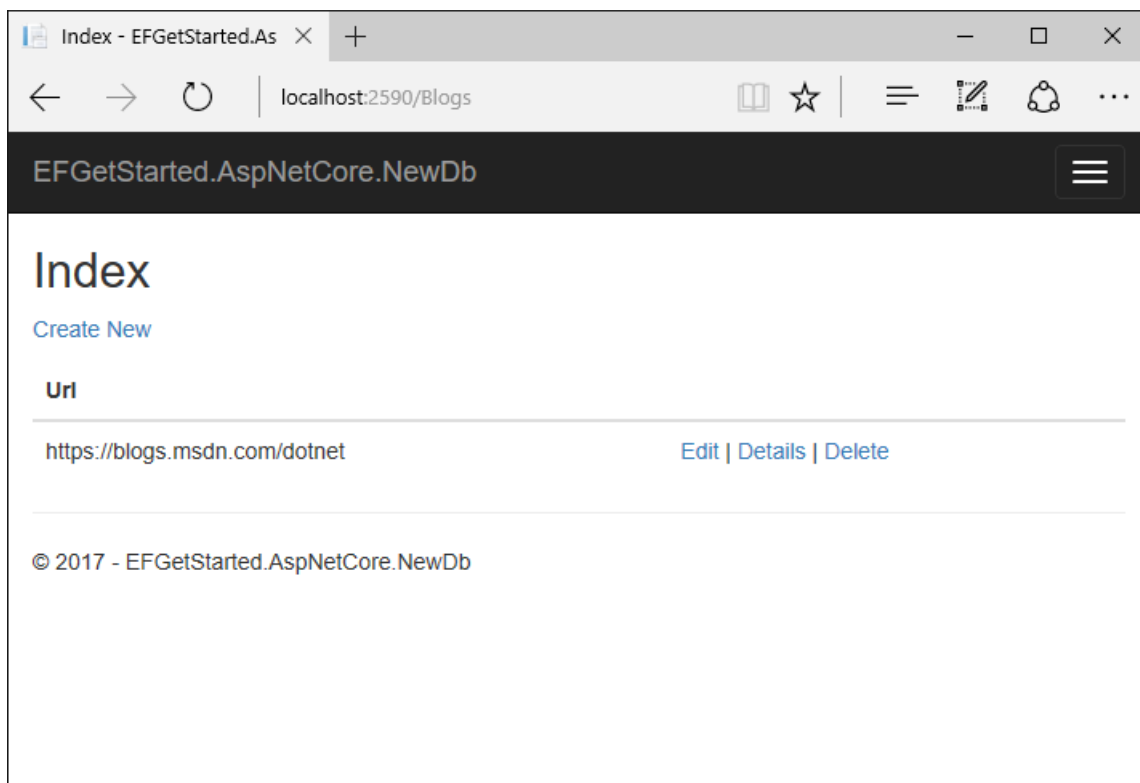
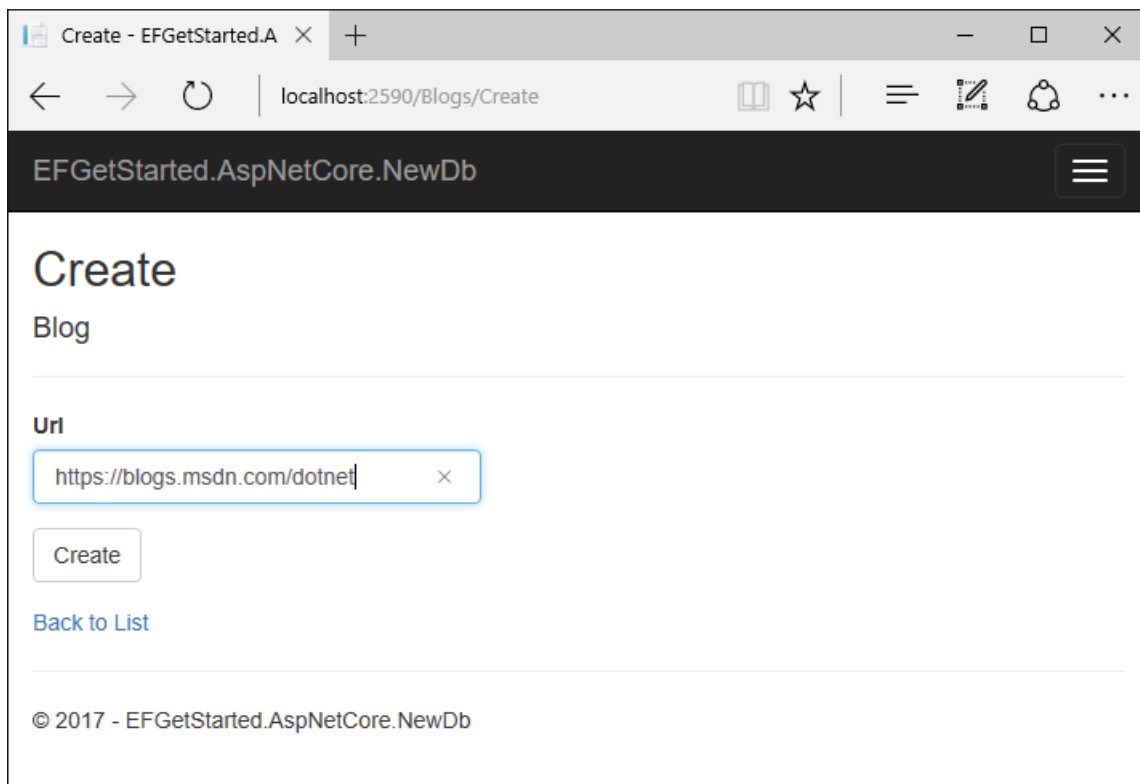
现在基架已启用，我们可以为 `Blog` 实体搭建一个控制器。

- 在“解决方案资源管理器”中，右键单击“控制器”文件夹，然后选择“添加”>“控制器”。
- 选择“视图使用 Entity Framework 的 MVC 控制器”，然后单击“确定”。
- 将“模型类”设置为“Blog”，将“数据上下文类”设置为“BlogggingContext”。
- 单击 **添加**。

## 运行此应用程序

按 F5 以运行和测试应用程序。

- 导航到 `/Blogs`
- 使用创建链接创建一些博客条目。测试“详细信息”和“删除”链接。



## 其他资源

- [EF - 通过 SQLite 新建数据库](#) - 跨平台的控制台 EF 教程。
- [Mac 或 Linux 上的 ASP.NET Core MVC 简介](#)
- [具有 Visual Studio 的 ASP.NET Core MVC 简介](#)
- [借助 Visual Studio 使用 ASP.NET Core 和 Entity Framework Core 入门](#)

# 通过现有数据库在 ASP.NET Core 上开始使用 EF Core

2018/3/25 • 6 min to read • [Edit Online](#)

在本演练中，你将使用 Entity Framework 构建执行基本数据访问的 ASP.NET Core MVC 应用程序。你将使用反向工程基于现有数据库创建 Entity Framework 模型。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 系统必备

完成本演练需要以下先决条件：

- 具有以下工作负载的 [Visual Studio 2017 15.3](#)：
  - “ASP.NET 和 Web 开发”（位于“Web 和云”下）
  - “.NET Core 跨平台开发”（位于“其他工具集”下）
- [.NET Core 2.0 SDK](#).
- [博客数据库](#)

### 博客数据库

本教程使用 LocalDb 实例上的博客数据库作为现有数据库。

## 提示

如果你已在其他教程中创建了博客数据库，则可以跳过这些步骤。

- 打开 Visual Studio
- “工具”->“连接到数据库...”
- 选择“Microsoft SQL Server”，然后单击“继续”
- 输入“(localdb)\mssqllocaldb”作为服务器名称
- 输入“master”作为数据库名称，然后单击“确定”
- Master 数据库现在显示在“服务器资源管理器”的“数据连接”中
- 右键单击“服务器资源管理器”中的数据库，然后选择“新建查询”
- 将下面列出的脚本复制到查询编辑器中
- 右键单击查询编辑器，然后选择“执行”

```

CREATE DATABASE [Blogging];
GO

USE [Blogging];
GO

CREATE TABLE [Blog] (
    [BlogId] int NOT NULL IDENTITY,
    [Url] nvarchar(max) NOT NULL,
    CONSTRAINT [PK_Blog] PRIMARY KEY ([BlogId])
);
GO

CREATE TABLE [Post] (
    [PostId] int NOT NULL IDENTITY,
    [BlogId] int NOT NULL,
    [Content] nvarchar(max),
    [Title] nvarchar(max),
    CONSTRAINT [PK_Post] PRIMARY KEY ([PostId]),
    CONSTRAINT [FK_Post_Blog_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blog] ([BlogId]) ON DELETE CASCADE
);
GO

INSERT INTO [Blog] (Url) VALUES
('http://blogs.msdn.com/dotnet'),
('http://blogs.msdn.com/webdev'),
('http://blogs.msdn.com/visualstudio')
GO

```

## 创建新项目

- 打开 Visual Studio 2017
- “文件”>“新建”>“项目...”
- 从左侧菜单中选择“已安装”->“模板”->“Visual C#”->“Web”
- 选择“ASP.NET Core Web 应用程序(.NET Core)”项目模板
- 输入 **EFGetStarted.AspNetCore.ExistingDb** 作为名称，然后单击“确定”
- 等待“新建 ASP.NET Core Web 应用程序”对话框显示出来
- 在“ASP.NET Core 模板 2.0”下，选择“Web 应用程序(模型视图控制器)”
- 确保将“身份验证”设置为“无身份验证”
- 单击“确定”

## 安装 Entity Framework

要使用 EF Core，请为要作为目标对象的数据库提供程序安装程序包。本演练使用 SQL Server。有关可用提供程序的列表，请参阅[数据库提供程序](#)。

- “工具”>“NuGet 包管理器”>“包管理器控制台”
- 运行 `Install-Package Microsoft.EntityFrameworkCore.SqlServer`

我们将使用一些 Entity Framework Tools 从数据库创建模型。因此，我们也会安装此工具包：

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

我们稍后将使用一些 ASP.NET Core 基架工具来创建控制器和视图。因此，我们也会安装此设计包：

- 运行 `Install-Package Microsoft.VisualStudio.Web.CodeGeneration.Design`

# 对模型实施反向工程

现在是时候基于现有数据库创建 EF 模型了。

- “工具”->“NuGet 包管理器”->“包管理器控制台”
- 运行以下命令以从现有数据库创建模型：

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

如果收到错误，指出 `The term 'Scaffold-DbContext' is not recognized as the name of a cmdlet`，请关闭并重新打开 Visual Studio。

## 提示

可以通过将 `-Tables` 参数添加到上述命令来指定要为哪些表生成实体。例如，`-Tables Blog,Post`。

反向工程过程基于现有数据库的架构创建实体类（`Blog.cs` & `Post.cs`）和派生上下文（`BloggingContext.cs`）。

实体类是简单的 C# 对象，代表要查询和保存的数据。

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

namespace EFGetStarted.AspNetCore.ExistingDb.Models
{
    public partial class Blog
    {
        public Blog()
        {
            Post = new HashSet<Post>();
        }

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

        public virtual ICollection<Post> Post { get; set; }
    }
}
```

上下文表示与数据库的会话，并允许查询和保存实体类的实例。

```

public partial class BloggingContext : DbContext
{
    public virtual DbSet<Blog> Blog { get; set; }
    public virtual DbSet<Post> Post { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        if (!optionsBuilder.IsConfigured)
        {
            #warning To protect potentially sensitive information in your connection string, you should move it
            out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection
            strings.
            optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;");
        }
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>(entity =>
        {
            entity.Property(e => e.Url).IsRequired();
        });

        modelBuilder.Entity<Post>(entity =>
        {
            entity.HasOne(d => d.Blog)
                .WithMany(p => p.Post)
                .HasForeignKey(d => d.BlogId);
        });
    }
}

```

## 通过依赖关系注入注册上下文

依赖关系注入的概念是 ASP.NET Core 的核心。服务(例如 `BloggingContext`)在应用程序启动期间通过依赖关系注入进行注册。然后, 通过构造函数参数或属性为需要这些服务的组件(如 MVC 控制器)提供相应服务。有关依赖关系注入的详细信息, 请参阅 ASP.NET 网站上的文章[依赖关系注入](#)。

### 删除内联上下文配置

在 ASP.NET Core 中, 通常在 `Startup.cs` 中进行配置。为符合这种模式, 我们将把数据库提供程序的配置移至 `Startup.cs`。

- 打开 `Models\BloggingContext.cs`
- 删除 `OnConfiguring(...)` 方法

```

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    #warning To protect potentially sensitive information in your connection string, you should move it out of
    source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.
    optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;");
}

```

- 添加以下构造函数, 这将允许通过依赖关系注入将配置传递到上下文中

```

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

```

## 在 Startup.cs 中注册并配置上下文

为了使 MVC 控制器能够使用 `BloggingContext`，我们将把它注册为一项服务。

- 打开 Startup.cs
- 在文件开头添加以下 `using` 语句

```
using EFGetStarted.AspNetCore.ExistingDb.Models;  
using Microsoft.EntityFrameworkCore;
```

现在我们可以使用 `AddDbContext(...)` 方法将其注册为服务。

- 找到 `ConfigureServices(...)` 方法
- 添加以下代码以将上下文注册为服务

```
// This method gets called by the runtime. Use this method to add services to the container.  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddMvc();  
  
    var connection = @"Server=  
(localdb)\mssqllocaldb;Database=Blogging;Trusted_Connection=True;ConnectRetryCount=0";  
    services.AddDbContext<BloggingContext>(options => options.UseSqlServer(connection));  
}
```

### 提示

在实际的应用程序中，通常会将连接字符串置于配置文件中。为了简单起见，我们在代码中对它进行定义。有关详细信息，请[参阅连接字符串](#)。

## 创建控制器

接下来，我们将在项目中启用基架。

- 在“解决方案资源管理器”中，右键单击“控制器”文件夹，然后选择“添加”->“控制器...”
- 选择“全部依赖项”，然单击“添加”
- 你可以忽略随后打开的 `ScaffoldingReadMe.txt` 文件中的说明

现在基架已启用，我们可以为 `Blog` 实体搭建一个控制器。

- 在“解决方案资源管理器”中，右键单击“控制器”文件夹，然后选择“添加”->“控制器...”
- 选择“视图使用 Entity Framework 的 MVC 控制器”，然后单击“确定”
- 将“模型类”设置为“Blog”，将“数据上下文类”设置为“BloggingContext”
- 单击“添加”

## 运行此应用程序

现在可以运行应用程序来查看其实际运行情况。

- “调试”->“开始执行(不调试)”
- 应用程序将生成并在 web 浏览器中打开
- 导航到 `/Blogs`
- 单击“新建”
- 输入新博客的 Url，然后单击“创建”

Create - EFGetStarted.A

+

localhost:2590/Blogs/Create

☆

≡

...

EFGetStarted.AspNetCore.NewDb

# Create

## Blog

Url

https://blogs.msdn.com/dotnet

Create

[Back to List](#)

© 2017 - EFGetStarted.AspNetCore.NewDb

Index - EFGetStarted.As

+

localhost:2590/Blogs

☆

≡

...

EFGetStarted.AspNetCore.NewDb

# Index

[Create New](#)

Url	
http://blogs.msdn.com/dotnet	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/webdev	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/visualstudio	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/dotnet	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/webdev	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/visualstudio	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/dotnet	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/webdev	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
http://blogs.msdn.com/visualstudio	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2017 - EFGetStarted.AspNetCore.ExistingDb



# 基于通用 Windows 平台 (UWP) 的 EF Core 入门

2018/5/17 • 1 min to read • [Edit Online](#)

无需具备 Entity Framework Core 或 Visual Studio 背景知识即可学习这 101 个教程。这些教程会逐步演示如何创建简单的通用 Windows 平台 (UWP) 应用程序, 以便查询和保存数据库中的数据。

可在 [developer.microsoft.com/windows/apps/develop](https://developer.microsoft.com/windows/apps/develop) 中找到 UWP 文档。

# 通过新数据库在通用 Windows 平台 (UWP) 上开始使用 EF Core

2018/1/23 • 7 min to read • [Edit Online](#)

## 注意

本教程使用 EF Core 2.0.1 (与 ASP.NET Core 和 .NET Core SDK 2.0.3 一起发布)。EF Core 2.0.0 缺少一些重要的 bug 修复, 因而无法提供良好的 UWP 体验。

在本演练中, 你将构建一个通用 Windows 平台 (UWP) 应用程序, 该程序使用 Entity Framework 对本地 SQLite 数据库执行基本数据访问。

## 重要事项

考虑在针对 UWP 的 LINQ 查询中避免使用匿名类型。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。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 先决条件

完成本演练需要具有以下各项:

- [Windows 10 Fall Creators Update](#) (10.0.16299.0)
- [.NET Core 2.0.0 SDK](#) 或更高版本。
- 具有通用 Windows 平台开发工作负载的 [Visual Studio 2017](#) 15.4 版或更高版本。

## 创建新模型项目

### 警告

由于 .NET Core 工具与 UWP 项目交互的方式受到限制, 因此该模型需要放在非 UWP 项目中才能在包管理器控制台中运行迁移命令

- 打开 Visual Studio
- “文件”>“新建”>“项目...”
- 从左侧菜单中选择“模板”>“Visual C#”
- 选择“类库(.NET Standard)”项目模板。
- 为项目提供名称, 然后单击“确定”

## 安装 Entity Framework

要使用 EF Core, 请为要作为目标对象的数据库提供程序安装程序包。本演练使用 SQLite。有关可用提供程序的列表, 请参阅[数据库提供程序](#)。

- “工具”>“NuGet 包管理器”>“包管理器控制台”
- 运行 `Install-Package Microsoft.EntityFrameworkCore.Sqlite`

在本演练的后面部分, 我们还将一些 Entity Framework Tools 用于维护数据库。因此, 我们也会安装此工具包。

- 运行 `Install-Package Microsoft.EntityFrameworkCore.Tools`
- 编辑 .csproj 文件, 并将 `<TargetFramework>netstandard2.0</TargetFramework>` 替换为 `<TargetFrameworks>netcoreapp2.0;netstandard2.0</TargetFrameworks>`

## 创建你自己的模型

现在是时候定义构成模型的上下文和实体类了。

- “项目”>“添加类...”
- 输入“Model.cs”作为名称, 然后单击“确定”
- 将此文件的内容替换为以下代码

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

namespace EFGetStarted.UWP
{
    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");
        }
    }

    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; }
    }
}
```

## 创建新的 UWP 项目

- 打开 Visual Studio

- “文件”>“新建”>“项目...”
- 从左侧菜单中选择“模板”>“Visual C#”>“Windows 通用”
- 选择“空白应用(通用 Windows)”项目模板
- 为项目提供名称，然后单击“确定”
- 将目标版本和最低版本至少设置为 `Windows 10 Fall Creators Update (10.0; build 16299.0)`

## 创建数据库

现在你已经有了模型，可以使用迁移为自己创建数据库。

- “工具”->“NuGet 包管理器”->“包管理器控制台”
- 选择作为默认项目的模型项目并将其设置为启动项目
- 运行 `Add-Migration MyFirstMigration` 来为迁移搭建基架，从而为模型创建一组初始表。

我们希望在运行该应用程序的设备上创建数据库，因此，我们将添加一些代码，以便在应用程序启动时将任何待定迁移应用到本地数据库。应用程序第一次运行时，该操作将为我们创建本地数据库。

- 在“解决方案资源管理器”中，右键单击“App.xaml”，然后选择“查看代码”。
- 将突出显示的 using 添加到文件的开头
- 添加突出显示的代码，以应用任何待定迁移

```
using Microsoft.EntityFrameworkCore;
using System;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;

namespace EFGetStarted.UWP
{
    /// <summary>
    /// Provides application-specific behavior to supplement the default Application class.
    /// </summary>
    sealed partial class App : Application
    {
        /// <summary>
        /// Initializes the singleton application object. This is the first line of authored code
        /// executed, and as such is the logical equivalent of main() or WinMain().
        /// </summary>
        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;

            using (var db = new BloggingContext())
            {
                db.Database.Migrate();
            }
        }

        /// <summary>
        /// Invoked when the application is launched normally by the end user. Other entry points
        /// will be used such as when the application is launched to open a specific file.
        /// </summary>
        /// <param name="e">Details about the launch request and process.</param>
        protected override void OnLaunched(LaunchActivatedEventArgs e)
        {
            //
        }
    }
}
```

```

    }
    #if DEBUG
        if (System.Diagnostics.Debugger.IsAttached)
        {
            this.DebugSettings.EnableFrameRateCounter = true;
        }
    #endif

    Frame rootFrame = Window.Current.Content as Frame;

    // Do not repeat app initialization when the Window already has content,
    // just ensure that the window is active
    if (rootFrame == null)
    {
        // Create a Frame to act as the navigation context and navigate to the first page
        rootFrame = new Frame();

        rootFrame.NavigationFailed += OnNavigationFailed;

        if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            //TODO: Load state from previously suspended application
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
    }

    if (e.PrelaunchActivated == false)
    {
        if (rootFrame.Content == null)
        {
            // When the navigation stack isn't restored navigate to the first page,
            // configuring the new page by passing required information as a navigation
            // parameter
            rootFrame.Navigate(typeof(MainPage), e.Arguments);
        }
        // Ensure the current window is active
        Window.Current.Activate();
    }
}

/// <summary>
/// Invoked when Navigation to a certain page fails
/// </summary>
/// <param name="sender">The Frame which failed navigation</param>
/// <param name="e">Details about the navigation failure</param>
void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
{
    throw new Exception("Failed to load Page " + e.SourcePageType.FullName);
}

/// <summary>
/// Invoked when application execution is being suspended. Application state is saved
/// without knowing whether the application will be terminated or resumed with the contents
/// of memory still intact.
/// </summary>
/// <param name="sender">The source of the suspend request.</param>
/// <param name="e">Details about the suspend request.</param>
private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();
    //TODO: Save application state and stop any background activity
    deferral.Complete();
}
}
}

```

### 提示

如果将来对模型进行更改，则可以使用 `Add-Migration` 命令为新迁移搭建基架，以便将相应更改应用到数据库。当应用程序启动时，任何待定迁移都将应用到每个设备上的本地数据库。

EF 使用数据库中的 `__EFMigrationsHistory` 表来跟踪哪些迁移已经应用到数据库。

## 使用模型

你现在可以使用模型执行数据访问。

- 打开 MainPage.xaml
- 添加下面突出显示的网页加载处理程序和 UI 内容

```
<Page
    x:Class="EFGetStarted.UWP.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:EFGetStarted.UWP"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Loaded="Page_Loaded">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel>
            <TextBox Name="NewBlogUrl"></TextBox>
            <Button Click="Add_Click">Add</Button>
            <ListView Name="Blogs">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding Url}" />
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackPanel>
    </Grid>
</Page>
```

现在我们将添加代码来连接 UI 和数据库

- 在“解决方案资源管理器”中，右键单击“MainPage.xaml”，然后选择“查看代码”。
- 从以下列表中添加突出显示的代码

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409

namespace EFGetStarted.UWP
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }

        private void Page_Loaded(object sender, RoutedEventArgs e)
        {
            using (var db = new BloggingContext())
            {
                Blogs.ItemsSource = db.Blogs.ToList();
            }
        }

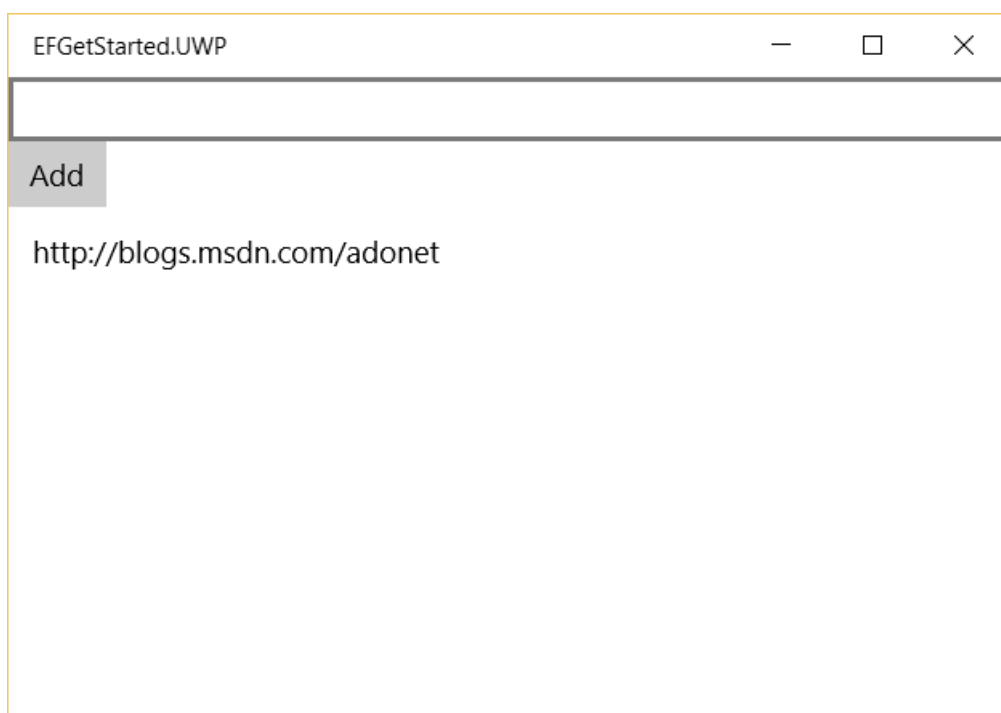
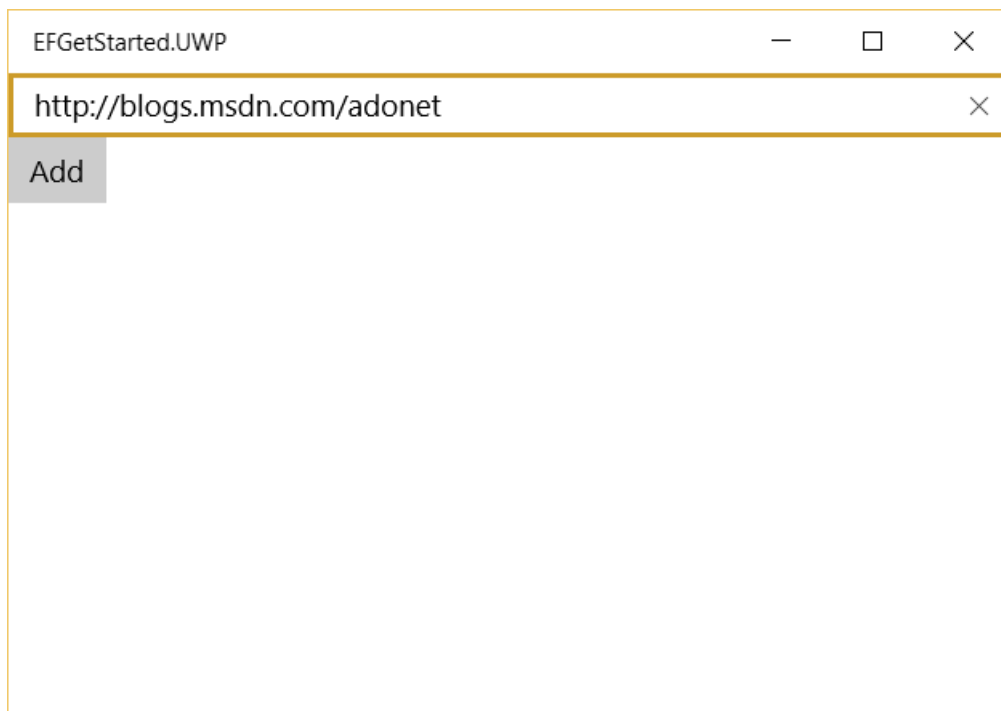
        private void Add_Click(object sender, RoutedEventArgs e)
        {
            using (var db = new BloggingContext())
            {
                var blog = new Blog { Url = NewBlogUrl.Text };
                db.Blogs.Add(blog);
                db.SaveChanges();

                Blogs.ItemsSource = db.Blogs.ToList();
            }
        }
    }
}

```

现在可以运行应用程序来查看其实际运行情况。

- “调试”>“开始执行(不调试)”
- 应用程序将生成并启动
- 输入 URL，然后单击“添加”按钮



## 后续步骤

### 提示

通过实现实体类型中的 `INotifyPropertyChanged`、`INotifyPropertyChanging`、`INotifyCollectionChanged` 并使用 `ChangeTrackingStrategy.ChangingAndChangedNotifications` 可改进 `SaveChanges()` 性能。

哇哦！你现在已经有了一个运行 Entity Framework 的简单 UWP 应用了。

查阅本文档中的其他文章，详细了解关于 Entity Framework 功能的信息。



# 创建模型

2018/5/3 • 1 min to read • [Edit Online](#)

Entity Framework 使用一组约定基于实体类的形状构建模型。可指定其他配置补充和/或替代由约定发现的内容。

本文介绍可应用于面向任何数据存储的模型的配置，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。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 配置方法

### Fluent API

可在派生上下文中替代 `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。

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

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

### 数据注释

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 配置覆盖。

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

# 包括和排除类型

2018/1/23 • 1 min to read • [Edit Online](#)

在模型意味着，EF 的元数据，该类型并且将尝试读取和写入从/到数据库的实例中包含一种类型。

## 约定

按照约定，在中公开的类型 `DbSet` 上您的上下文属性都包含在您的模型。此外，会在中提到的类型 `OnModelCreating`，还提供了方法。最后，通过以递归方式浏览发现的类型的导航属性找到的任何类型也会包括在模型中。

例如，下面的代码清单中发现所有三种类型：

- `Blog` 因为在公开 `DbSet` 上下文属性
- `Post` 因为发现通过 `Blog.Posts` 导航属性
- `AuditEntry` 因为中提到 `OnModelCreating`

```
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; }
}
```

## 数据注释

你可以使用数据注释从模型排除类型。

```

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

    public BlogMetadata Metadata { get; set; }
}

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

```

## Fluent API

Fluent API 可用于从模型中排除类型。

```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}

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

    public BlogMetadata Metadata { get; set; }
}

public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}

```

# 包括和排除属性

2018/1/23 • 1 min to read • [Edit Online](#)

在模型中的属性包括意味着 EF 具有该属性有关的元数据，并且将尝试读取和写入从/到数据库的值。

## 约定

按照约定，将在模型中包括具有 getter 和 setter 的公共属性。

## 数据注释

可以使用数据注释从模型中排除属性。

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

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

## Fluent API

可以使用 Fluent API 若要从模型中排除属性。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Ignore(b => b.LoadedFromDatabase);
    }
}

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

    public DateTime LoadedFromDatabase { get; set; }
}
```

# 密钥（主）

2018/1/23 • 1 min to read • [Edit Online](#)

一个键用作每个实体实例的主唯一标识符。使用关系数据库时此方法映射到的概念 *主键*。你还可以配置不是主键的唯一标识符 (请参阅[备用键](#)有关详细信息)。

## 约定

按照约定, 属性名为 `Id` 或 `<type name>Id` 将配置为实体的键。

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

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

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

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

## 数据注释

数据注释可用于配置要作为键的实体的单个属性。

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

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

## Fluent API

Fluent API 可用于配置要作为键的实体的单个属性。

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

class Car
{
    public string LicensePlate { get; set; }

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

```

Fluent API 还可用于配置多个属性，以作为键的实体（称为复合键）。复合键仅可以配置即可使用 Fluent API-约定将永远不会在安装程序组合键，不可以使用数据注释若要配置一个。

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

class Car
{
    public string State { get; set; }
    public string LicensePlate { get; set; }

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

```

# 生成的值

2018/5/3 • 5 min to read • [Edit Online](#)

## 值生成模式

有三种可用于属性的值生成模式。

### 没有值生成

没有值生成意味着，你将始终提供有效的值保存到数据库。在它们添加到上下文之前，必须将此有效的值分配到新的实体。

### 在生成的值添加

在生成的值添加表示为新的实体生成一个值。

根据正在使用的数据库的提供程序的情况下，值可能生成由 EF 或数据库中的客户端。如果值由数据库生成的 EF 可能分配一个临时值，当将实体添加到上下文。然后在过程的数据库生成值替换为此临时值 `SaveChanges()`。

如果向已分配给属性的值的上下文中添加实体，然后 EF 将尝试插入该值，而不是生成一个新。属性将被视为已分配如果未分配的 CLR 的默认值的值（`null` 为 `string`，`0` 为 `int`，`Guid.Empty` 为 `Guid` 等。）。有关详细信息，请参阅[生成的属性的显式值](#)。

#### 警告

如何为添加的实体生成值将取决于正在使用的数据库提供程序。数据库提供程序可能在安装程序自动值生成对于某些属性类型，但其他可能需要你进行手动设置值的生成方式。

例如，在使用 SQL Server 时，值将为自动生成 `GUID`（使用 SQL Server 顺序 GUID 算法）的属性。但是，如果你指定 `DateTime` 生成属性上添加，则您必须先设置要生成的值的方法。执行此操作，一种方法是配置的默认值为 `GETDATE()`，请参阅[默认值](#)。

### 生成的值在添加或更新

生成的值在添加或更新意味着每次（插入或更新），将保存该记录生成了的新值。

如 `value generated on add`，如果在新添加的实体，将而不是所生成的值插入值实例上指定属性的值。还有可能要设置一个显式值更新时。有关详细信息，请参阅[生成的属性的显式值](#)。

## 警告

如何添加和已更新的实体生成值将取决于正在使用的数据库提供程序。数据库提供程序可能在安装程序自动值生成对于某些属性类型, 而其他人将需要你手动设置值的生成方式。

例如, 在使用 SQL Server, `byte[]` 生成上设置的属性添加或更新和标记为并发标记将与设置 `rowversion` 数据类型-以便将数据库中生成值。但是, 如果你指定 `DateTime` 生成属性上添加或更新, 则您必须先设置要生成的值的方法。执行此操作, 一种方法是配置的默认值为 `GETDATE()` (请参阅[默认值](#)) 来生成新行的值。然后可以使用数据库触发器在(如下面的示例触发器)的更新过程中生成值。

```
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
```

## 约定

按照约定, 非复合主键的类型 `short`、`int`、`long`、或 `Guid` 将安装程序能够生成上添加的值。所有其他属性将与不值生成的安装程序。

## 数据注释

### 没有值生成 (数据注释)

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

### 在生成的值添加 (数据注释)

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

## 警告

这只是让 EF 知道为添加的实体生成值, 它不保证 EF 将设置生成值的实际机制。请参阅[上生成的值添加](#)有关详细信息部分。



## 在生成的值将添加或更新（数据注释）

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

### 警告

这只是让 EF 知道值生成的添加或更新实体, 它不保证 EF 将设置生成值的实际机制。请参阅[上生成的值添加或更新](#)有关详细信息部分。

## Fluent API

Fluent API 可用于更改给定属性的值生成模式。

### 没有值生成 (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.BlogId)
    .ValueGeneratedNever();
```

### 在生成的值添加 (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.Inserted)
    .ValueGeneratedOnAdd();
```

### 警告

`ValueGeneratedOnAdd()` 只是让 EF 知道为添加的实体生成值, 它不保证 EF 将设置生成值的实际机制。请参阅[上生成的值添加](#)有关详细信息部分。

### 在生成的值将添加或更新 (Fluent API)

```
modelBuilder.Entity<Blog>()
    .Property(b => b.LastUpdated)
    .ValueGeneratedOnAddOrUpdate();
```

### 警告

这只是让 EF 知道值生成的添加或更新实体, 它不保证 EF 将设置生成值的实际机制。请参阅[上生成的值添加或更新](#)有关详细信息部分。

# 必选和可选属性

2018/1/23 • 1 min to read • [Edit Online](#)

该属性被视为可选如果有效它才能包含 `null`。如果 `null` 不是有效的值分配给属性中，则可以认为是必需的属性。

## 约定

按照约定，其 CLR 类型可以包含 `null` 的属性将被配置为可选（`string`，`int?`，`byte[]` 等。）。将配置其 CLR 类型不能包含 `null` 的属性所需的方式（`int`，`decimal`，`bool` 等。）。

### 注意

其 CLR 类型不能包含 `null` 的属性不能配置为可选。始终将所需的实体框架视为属性。

## 数据注释

可以使用数据注释以指示需要一个属性。

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

## Fluent API

可以使用 Fluent API 以指示需要一个属性。

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

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

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

# 最大长度

2018/1/23 • 1 min to read • [Edit Online](#)

配置的最大长度提供了关于给定属性适用于使用适当的数据类型与数据存储的提示。最大长度仅适用于数组数据类型, 如 `string` 和 `byte[]`。

## 注意

实体框架不执行最大长度的任何的验证, 然后再将数据传递给提供程序。负责要验证如果相应的提供程序或数据存储区。例如, 面向 SQL Server, 超过最大长度将结果作为基础列的数据类型引发异常时将不允许过多的数据存储。

## 约定

按照约定, 它将保持最多要选择适当的数据类型的属性的数据库提供程序。对于具有长度的属性, 数据库提供程序通常将选择的数据类型必须允许的最长长度的数据。例如, Microsoft SQL Server 将使用 `nvarchar(max)` 为 `string` 属性 (或 `nvarchar(450)` 如果列用作键)。

## 数据注释

数据注释可用于配置属性的最大长度。在此示例中, 面向 SQL Server, 这将导致 `nvarchar(500)` 正在使用的数据类型。

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

## Fluent API

Fluent API 可用于配置属性的最大长度。在此示例中, 面向 SQL Server, 这将导致 `nvarchar(500)` 正在使用的数据类型。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasMaxLength(500);
    }
}

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

# 并发标记

2018/3/5 • 2 min to read • [Edit Online](#)

## 注意

本页介绍如何配置并发标记。请参阅[处理并发冲突](#)并发控制 EF 核心以及如何处理应用程序中的并发冲突的示例中的工作原理的详细说明。

属性配置为并发标记用于实现乐观并发控制。

## 约定

按照约定, 属性被永远不会配置为并发标记。

## 数据注释

你可以使用数据注释将属性配置为并发标记。

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

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

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

## Fluent API

Fluent API 可用于将属性配置为并发标记。

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .Property(p => p.LastName)
            .IsConcurrencyToken();
    }
}

public class Person
{
    public int PersonId { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
}
```

## 时间戳/行版本

时间戳是每次插入或更新行时，由数据库生成一个新值是其中一个属性。属性也将被视为并发标记。这可确保如果其他人进行了修改你尝试更新由于数据查询的行，则将收到异常。

这如何实现由正在使用的数据库提供程序。对于 SQL Server，时间戳通常使用上 *byte [] 属性，用于将设置为ROWVERSION数据库中的列。*

## 约定

按照约定，属性被永远不会配置为时间戳。

## 数据注释

数据注释可用于将属性配置为时间戳。

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

    public string Url { get; set; }

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

## Fluent API

Fluent API 可用于将属性配置为时间戳。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(p => p.Timestamp)
            .IsRowVersion();
    }
}

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

# 隐藏属性

2018/1/23 • 2 min to read • [Edit Online](#)

卷影属性是指你.NET 实体类中未定义但 EF 核心模型中该实体类型定义。值和状态的这些属性将保留只是在更改跟踪器。

不应在映射的实体类型公开的数据库中没有数据时，卷影属性非常有用。它们通常用于外键属性，其中两个实体之间的关系由在数据库中，某个外键值，但使用实体类型之间的导航属性的实体类型上托管关系。

卷影属性值可以获取和通过更改 `ChangeTracker` API。

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

可以通过 LINQ 查询中引用隐藏属性 `EF.Property` 静态方法。

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

## 约定

当发现一种关系，但没有外键属性找到依赖的实体类中，可以通过约定创建卷影属性。在这种情况下，将引入了卷影的外键属性。将名为卷影外键属性 `<navigation property name><principal key property name>`（依赖的实体，用于指向主体实体，对导航用于命名）。如果主体键属性名称中包含的名称的导航属性，则名称只需将 `<principal key property name>`。如果依赖实体中没有任何导航属性，则主体类型名称使用在其位置。

例如，下面的代码清单将导致 `BlogId` 引入到的卷影属性 `Post` 实体。

```
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; }

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

## 数据注释

使用数据注释，可以创建卷影属性。

# Fluent API

Fluent API 可用于配置卷影属性。一旦调用了字符串重载的 `Property` 你可以将任何其他属性将在配置调用的链接。

如果名称提供给 `Property` 方法与匹配的名称的现有属性（卷影属性或其中一个实体类上定义），则代码将配置该现有属性，而不是引入一个新的卷影属性。

```
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; }
}
```

# 关系

2018/1/23 • 16 min to read • [Edit Online](#)

关系定义了两个实体将互相关联起来。在关系数据库中，它被表示通过外键约束。

## 注意

大部分这篇文章中的示例使用一个对多关系来演示概念。有关一对一和多对多关系的示例，请参阅[其他关系模式](#)文章结尾部分。

## 术语的定义

有大量用于描述关系的术语

- **依赖的实体**: 这是包含外键属性的实体。有时称为 child 的关系。
- **主体实体**: 这是包含主/备用键属性的实体。有时称为 'parent' 的关系。
- **外键**: 中用于存储与相关实体的主体键属性的值的依赖实体属性。
- **主体的密钥**: 唯一标识的主体实体属性。这可能是 primary key 或备用密钥。
- **导航属性**: 包含对相关实体引用的主体和/或依赖实体上定义的属性。
  - **集合导航属性**: 一个导航属性, 包含对多个相关实体的引用。
  - **引用导航属性**: 一个导航属性, 保存对单个相关实体的引用。
  - **反向导航属性**: 在讨论特定导航属性时, 此术语是指关系另一端的导航属性。

下面的代码清单显示之间的一个对多关系 `Blog` 和 `Post`

- `Post` 是依赖实体
- `Blog` 是主体实体
- `Post.BlogId` 是外键
- `Blog.BlogId` 是（在这种情况下它是主键, 而不是备用密钥）的主体键
- `Post.Blog` 是引用导航属性
- `Blog.Posts` 是集合导航属性
- `Post.Blog` 是的反向导航属性 `Blog.Posts`（反之亦然）



```

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; }
}

```

## 约定

按照约定, 发现的类型上的导航属性时, 将创建关系。如果它指向的类型不能将映射为标量类型由当前的数据库提供程序, 该属性被视为一个导航属性。

### 注意

关系发现的约定将始终以目标主体实体的主键。若要针对备用密钥, 必须使用 Fluent API 执行其他配置。

### 完全定义的关系

关系的最常见模式是能够在两端的关系和依赖的实体类中定义的外键属性上定义的导航属性。

- 如果两个类型之间找到一对导航属性, 则它们将可配置为位于同一关系的反向导航属性。
- 如果依赖实体包含名为的属性 `<primary key property name>`, `<navigation property name><primary key property name>`, 或 `<principal entity name><primary key property name>` 然后会将其配置为外键。

```

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; }
}

```

### 警告

如果有多个定义两个类型之间的导航属性 (即指向对方的导航的多个非重复对)、约定将然后创建任何关系, 并且将需要手动配置它们以确定如何导航属性对长达。

## 没有外键属性

尽管建议具有外键属性的依赖的实体类中定义，则不需要。如果不找到任何外键属性，则卷影的外键属性，将名称引入 `<navigation property name><principal key property name>` (请参阅[隐藏属性](#)有关详细信息)。

```
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; }
}
```

## 一个导航属性

包括一个导航属性（没有反向导航和没有外键属性）已足够具有关系定义的约定。你还可以单个导航属性和外键属性。

```
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; }
}
```

## 级联删除

按照约定，级联删除将设置为*Cascade*的所需的的关系和*ClientSetNull*可选关系。*级联*意味着依赖实体也会被删除。*ClientSetNull*意味着不会加载到内存的依赖实体仍保持不变并且必须是手动将其删除，或更新以指向有效的主体实体。对于加载到内存的实体，EF 核心将尝试将外键属性设置为 null。

请参阅[必选和可选关系](#)部分以了解必需和可选关系之间的差异。

请参阅[级联删除](#)更多详细信息的不同删除行为和约定所使用的默认值。

## 数据注释

有两个用于配置关系的数据注释 `[ForeignKey]` 和 `[InverseProperty]`。

### [ForeignKey]

数据注释可用于配置哪些属性应用作给定关系的外键属性。这通常是约定没有发现外键属性。

```

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; }

    public int BlogForeignKey { get; set; }

    [ForeignKey("BlogForeignKey")]
    public Blog Blog { get; set; }
}

```

#### 提示

`[ForeignKey]` 批注可以放置在关系中这两个导航属性。它不需要继续依赖的实体类中的导航属性。

## [InverseProperty]

数据注释可用于配置如何依赖和主体实体的导航属性配对。这通常是多个对两个实体类型之间的导航属性时。

```

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

    public int AuthorUserId { get; set; }
    public User Author { get; set; }

    public int ContributorUserId { get; set; }
    public User Contributor { get; set; }
}

public class User
{
    public string UserId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    [InverseProperty("Author")]
    public List<Post> AuthoredPosts { get; set; }

    [InverseProperty("Contributor")]
    public List<Post> ContributedToPosts { get; set; }
}

```

## Fluent API

若要配置关系 Fluent API 中, 启动时通过标识构成关系的导航属性。 `HasOne` 或 `HasMany` 标识你正在开始配置的实体类型上的导航属性。然后链接到调用 `WithOne` 或 `WithMany` 来标识反向导航。 `HasOne / WithOne` 用于引用导航属性和 `HasMany / WithMany` 用于集合导航属性。

```

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

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

```

## 一个导航属性

如果你只有一个导航属性，则存在一些无参数重载 `WithOne` 和 `WithMany`。这表示没有从概念上讲引用或集合上的另一端的关系，但不未包含在实体类中任何导航属性。

```

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

Fluent API 可用于配置哪些属性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; }
}
```

以下代码列表演示如何配置的组合外键。

```

class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

        modelBuilder.Entity<RecordOfSale>()
            .HasOne(s => s.Car)
            .WithMany(c => c.SaleHistory)
            .HasForeignKey(s => new { s.CarState, s.CarLicensePlate });
    }
}

public class Car
{
    public string State { 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 CarState { get; set; }
    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}

```

你可以使用的字符串重载 `HasForeignKey(...)` 卷影属性配置为外键 (请参阅[隐藏属性](#)有关详细信息)。建议显式将卷影属性添加到模型, 然后再使用它作为外键 (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; }
}

```

## 主体密钥

如果你想要引用以外的主键属性的外键，可以使用 Fluent API 来配置该关系的主体键属性。自动配置为该主体的密钥将该属性是作为备用键的安装程序 (请参阅[备用键](#)有关详细信息)。

```

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; }
}

```

以下代码列表演示如何配置主体组合键。



```

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 => new { s.CarState, s.CarLicensePlate })
            .HasPrincipalKey(c => new { c.State, c.LicensePlate });
    }
}

public class Car
{
    public int CarId { get; set; }
    public string State { 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 CarState { get; set; }
    public string CarLicensePlate { get; set; }
    public Car Car { get; set; }
}

```

### 警告

指定主体键属性的顺序必须与匹配外键指定的顺序。

## 必选和可选的关系

Fluent API 可用于配置关系为必需或可选。最终此参数控制外键属性是必需还是可选。使用卷影状态外键时，这是最有用。如果你有在实体类中的外键属性，则关系的 requiredness 确定基于外键属性是必需还是可选 (请参阅[必选和可选属性](#)的详细信息信息)。

```
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)
            .IsRequired();
    }
}

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; }
}
```

## 级联删除

Fluent API 可用于显式配置为某个给定的关系的级联删除行为。

请参阅[级联删除](#)上保存的数据部分中有关每个选项的详细讨论。

```

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)
            .OnDelete(DeleteBehavior.Cascade);
    }
}

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; }
}

```

## 其他关系模式

### 一对一

一对一关系两端具有引用导航属性。它们遵循相同的约定一个对多关系，但在外键属性，以确保只有一个依赖于与每个主体引入唯一索引。

```

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; }
}

```

#### 注意

EF 将选择一个要为基于其能够检测到外键属性的依赖项的实体。如果错误实体被选作依赖项，可以使用 Fluent API 要更正此问题。

如果配置使用 Fluent API 的关系, 则使用 `HasOne` 和 `WithOne` 方法。

配置你需要指定相关的实体类型中的外键时请注意提供的泛型参数 `HasForeignKey` 下面的列表中。一个对多关系中很明显的实体引用导航是依赖项和一个具有集合是的主体。但这不是这样一对一关系-因此中显式定义的需要。

```
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(p => p.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

尚不支持而不需要的实体类来表示联接表的多对多关系。但是, 您可以通过包括联接表和映射两个单独的一对多关系的实体类表示多对多关系。

```

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

2018/1/23 • 1 min to read • [Edit Online](#)

索引是跨多个数据存储的一个公共概念。尽管数据存储中的其实现可能会有所不同，它们用于使基于列（或一组列）的查找更加高效。

## 约定

按照约定，为外键使用每个属性（或组的属性）中创建索引。

## 数据注释

不使用数据注释创建索引。

## Fluent API

Fluent API 可用于的单个属性上指定索引。默认情况下，索引将非唯一。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url);
    }
}

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

此外可以指定索引应是唯一的这意味着任何两个实体可以具有的给定属性的相同值。

```
modelBuilder.Entity<Blog>()
    .HasIndex(b => b.Url)
    .IsUnique();
```

你还可以通过多个列中指定索引。

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>()
            .HasIndex(p => new { p.FirstName, p.LastName });
    }
}

public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

#### 提示

没有每个非重复的属性集只有一个索引。如果你使用 Fluent API 配置索引上的一组属性已索引定义, 方法是约定或以前的配置, 然后将会更改该索引的定义。这是在你想进一步配置的索引, 它由约定很有用。

# 备用键

2018/1/23 • 2 min to read • [Edit Online](#)

备用密钥用作每个实体实例的主键除了备用的唯一标识符。备用键可以用作关系的目标。使用关系数据库时此方法映射到备用键列和一个或多个外键约束引用的列上的唯一索引/约束的概念。

## 提示

如果你只是想要强制实施的列的唯一性, 则你希望唯一索引, 而不是备用密钥, 请参阅[索引](#)。在 EF, 备用键提供更强大的功能比唯一索引, 因为它们可以用作外键的目标。

备用键通常为你在需要时引入并不需要手动配置它们。请参阅[约定](#)有关详细信息。

## 约定

按照约定, 备用密钥时标识属性, 作为一种关系的目标不是主键, 为你引入。

```
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; }
}
```

## 数据注释

不能使用数据注释配置备用密钥。

## Fluent API



Fluent API 可用于配置单个属性用作备用键。

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

Fluent API 还可用于配置多个属性（称为复合的替换密钥）用作备用键。

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

class Car
{
    public int CarId { get; set; }
    public string State { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

# 继承

2018/1/23 • 1 min to read • [Edit Online](#)

EF 模型中的继承用于控制如何在数据库中表示的实体类中的继承。

## 约定

按照约定，由数据库提供程序，以确定继承在数据库中的表示方式。请参阅[继承 \(关系数据库\)](#)这是如何处理与关系数据库提供程序。

如果模型中显式包含两个或多个继承的类型，EF 将仅设置继承。EF 将不会扫描基表或派生类型，否则不包括在模型中。你可以在模型中包含类型，通过公开 *DbSet* 继承层次结构中的每个类型。

```
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; }
}
```

如果你不想要公开 *DbSet* 对于层次结构中的一个或多个实体，你可以使用 Fluent API 以确保它们包括在模型中。如果您不依赖于约定可以指定使用显式的基类型和 `HasBaseType`。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<RssBlog>().HasBaseType<Blog>();
    }
}
```

### 注意

你可以使用 `.HasBaseType((Type)null)` 以层次结构中删除的实体类型。

## 数据注释

数据注释不能用于配置继承。

## Fluent API

用于继承的 Fluent API 取决于你使用的数据库提供程序。请参阅[继承（关系数据库）](#)您可以对关系数据库提供程序执行的配置。

# 支持字段

2018/1/23 • 3 min to read • [Edit Online](#)

## 注意

此功能是 EF 核心 1.1 中的新增功能。

支持字段允许 EF 读取和/或写入一个字段，而不是一个属性。这一点可能很有用，当在类中的封装用于限制的使用和/或增强围绕访问数据的语义由应用程序代码，但值应进行读取和/或写入到数据库而不使用这些限制时 /增强功能。

## 约定

按照约定，将作为备份为给定属性（按优先顺序列出）的字段发现以下字段。字段只会发现的模型中包含的属性。在模型中包含属性的详细信息，请参阅[包括和排除属性](#)。

- `_<camel-cased property name>`
- `_<property name>`
- `m_<camel-cased property name>`
- `m_<property name>`

```
public class Blog
{
    private string _url;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _url; }
        set { _url = value; }
    }
}
```

当配置的支持字段时，EF 将直接写入该字段具体化数据库（而不是使用属性 setter）中的实体实例时。如果 EF 需要读取或写入在其他时间值，它将尽可能使用该属性。例如，如果 EF 需要更新属性的值，它将使用属性 setter，如果已定义。如果属性是只读的然后它将写入到字段。

## 数据注释

支持字段时，无法使用数据注释进行配置。

## Fluent API

Fluent API 可用于配置属性的支持字段。

```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .HasField("_validatedUrl");
    }
}

public class Blog
{
    private string _validatedUrl;

    public int BlogId { get; set; }

    public string Url
    {
        get { return _validatedUrl; }
    }

    public void SetUrl(string url)
    {
        using (var client = new HttpClient())
        {
            var response = client.GetAsync(url).Result;
            response.EnsureSuccessStatusCode();
        }

        _validatedUrl = url;
    }
}

```

## 控制当使用此字段

你可以配置当 EF 使用的字段或属性。请参阅[PropertyAccessMode 枚举](#)有关支持的选项。

```

modelBuilder.Entity<Blog>()
    .Property(b => b.Url)
    .HasField("_validatedUrl")
    .UsePropertyAccessMode(PropertyAccessMode.Field);

```

## 不带属性的字段

你还可以在实体类中，没有相应的 CLR 属性，但改为使用字段来存储实体中的数据在模型中创建概念的属性。这是不同于[隐藏属性](#)、数据更改跟踪器中的存储位置。此值通常将在实体类使用方法来获取/设置值使用。

可让 EF 中的字段的名称 `Property(...)` API。如果不没有具有给定名称的任何属性，将查找字段 EF。

```

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;
    }
}

```

你还可以选择为之外的字段名称的名称的属性。创建模型，然后使用此名称，最值得注意的是它将用于映射到数据库中的列名称。

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .Property<string>("Url")
        .HasField("_validatedUrl");
}

```

当实体类中有任何属性时，你可以使用 `EF.Property(...)` LINQ 查询来指代从概念上讲是模型的一部分的属性中的方法。

```

var blogs = db.blogs.OrderBy(b => EF.Property<string>(b, "Url"));

```

# 值的转换

2018/5/17 • 4 min to read • [Edit Online](#)

## 注意

此功能是在 EF 核心 2.1 的新增功能。

值转换器允许读取或写入数据库时要转换的属性值。此转换可以是某个值与另一个相同类型（例如，加密字符串）或从一种类型的值的另一种类型（例如，转换枚举值到和从数据库中的字符串。）的值

## 基础知识

值转换器指定方面 `ModelClrType` 和 `ProviderClrType`。模型类型是属性的中的实体类型的.NET 类型。提供程序类型是理解数据库提供程序的.NET 类型。例如，若要在数据库中的字符串的形式保存枚举，模型类型是一种枚举，和提供程序类型是 `String`。这两种类型可以是相同的。

使用两个定义转换 `Func` 表达式树：一个从 `ModelClrType` 到 `ProviderClrType` 和从其他 `ProviderClrType` 到 `ModelClrType`。使用表达式树，这样它们可以编译到数据库访问代码的高效转换。对于复杂的转换，表达式树可能简单调用执行转换的方法。

## 配置值转换器

中的你 `DbContext` `OnModelCreating` 属性定义值的转换。例如，考虑定义为枚举和实体类型：

```
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`（例如存储为字符串的枚举值定义转换"驴女子"，"帮凶"，...）在数据库中：

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Entity<Rider>()
        .Property(e => e.Mount)
        .HasConversion(
            v => v.ToString(),
            v => (EquineBeast)Enum.Parse(typeof(EquineBeast), v));
}
```

### 注意

A `null` 永远不会将该值传递到值转换器。这使转换的实现更容易, 并允许他们需要可以为 `null` 和不可为 `null` 属性之间共享。

## ValueConverter 类

调用 `HasConversion` 如上所示将创建 `ValueConverter` 实例并将其设置的属性上。 `ValueConverter` 改为显式创建。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, 这很有用。

### 注意

目前没有办法在一个位置中指定给定类型的每个属性, 必须使用相同的值转换器。此功能将考虑对将来的版本。

## 内置的转换器

EF 核心附带了一组预定义 `ValueConverter` 类, 在中找到 `Microsoft.EntityFrameworkCore.Storage.ValueConversion` 命名空间。这些是:

- `BoolToZeroOneConverter` -Bool 到零, 另一个
- `BoolToStringConverter` -Bool 为字符串, 如"Y"和"N"
- `BoolToTwoValuesConverter` -Bool 到任何两个值
- `BytesToStringConverter` 为 Base64 编码的字符串的字节数组
- `CastingConverter` -转换需要仅 Csharp 强制转换
- `CharToStringConverter` -Char 为单个字符的字符串
- `DateTimeOffsetToBinaryConverter` 的为二进制编码的 64 位值 DateTimeOffset
- `DateTimeOffsetToBytesConverter` 的指向字节数组 DateTimeOffset
- `DateTimeOffsetToStringConverter` -为字符串 DateTimeOffset
- `DateTimeToBinaryConverter` -为 64 位值包括 DateTimeKind DateTime
- `DateTimeToStringConverter` -从 DateTime 到 string
- `DateTimeToTicksConverter` 的为刻度 DateTime
- `EnumToNumberConverter` 的枚举基础数字
- `EnumToStringConverter` -为字符串枚举
- `GuidToBytesConverter` 的指向字节数组 Guid
- `GuidToStringConverter` 的为字符串 Guid
- `NumberToBytesConverter` -指向字节数组任何数字值
- `NumberToStringConverter` -为字符串任何数字值
- `StringToBytesConverter` -为 UTF8 字节字符串
- `TimeSpanToStringConverter` -时间跨度为字符串



- `TimeSpanToTicksConverter` 的为刻度时间跨度

请注意, `EnumToStringConverter` 包含在此列表。这意味着没有无需显式, 如上所示指定转换。相反, 只需使用内置的转换器:

```
var converter = new EnumToStringConverter<EquineBeast>();

modelBuilder
    .Entity<Rider>()
    .Property(e => e.Mount)
    .HasConversion(converter);
```

注意, 所有内置转换器是无状态, 因此单个实例可以安全地共享由多个属性。

## 预定义的转换

对于常见转换为其存在内置转换器没有无需显式指定转换器。相反, 只需配置应使用哪种提供程序类型和 EF 将自动使用适当的生成转换器。枚举字符串的转换用作上例中, 但 EF 将实际执行此操作自动如果配置的提供程序类型:

```
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操作。例如, 如果实体类型的定义, 如以便:

```
public class Rider
{
    public int Id { get; set; }

    [Column(TypeName = "nvarchar(24)")]
    public EquineBeast Mount { get; set; }
}
```

然后枚举值将另存为无需在 `OnModelCreating` 任何进一步配置数据库中的字符串。

## 限制

有几个已知的当前限制的值转换系统:

- 如上所述, `null` 无法转换。
- 目前没有办法分布到多个列或进行相反的转换的转换的一个属性。
- 使用的值转换可能会影响 EF 核心能够把表达式转换到 SQL。为这种情况下, 将记录警告。正在为未来版本考虑删除这些限制。

# 数据种子设定

2018/5/15 • 1 min to read • [Edit Online](#)

## 注意

此功能是在 EF 核心 2.1 的新增功能。

数据种子设定允许若要提供初始数据填充数据库。与不同在 ef6 更高版本, 在 EF 核心中设定数据种子都与关联模型配置的一部分作为实体类型。然后 EF 核心 [迁移](#) 什么插入、更新或删除操作需要时将数据库升级到新版本的模型的应用可以自动计算。

例如, 可用于此配置的种子数据 `Blog` 中 `OnModelCreating` :

```
modelBuilder.Entity<Blog>().HasData(new Blog {BlogId = 1, Url = "http://sample.com"});
```

若要添加实体的关系的外键值需要指定。经常外键属性是在卷影状态下, 因此, 若要能够设置匿名类的值应使用:

```
modelBuilder.Entity<Post>().HasData(  
    new {BlogId = 1, PostId = 1, Title = "First post", Content = "Test 1"},  
    new {BlogId = 1, PostId = 2, Title = "Second post", Content = "Test 2"});
```

一旦已添加实体, 则建议使用 [迁移](#) 以应用更改。

或者, 可以使用 `context.Database.EnsureCreated()` 以创建新的数据库包含种子数据, 例如对于一个测试数据库, 或者在使用内存中提供程序。请注意, 如果数据库已存在, `EnsureCreated()` 都将更新架构或数据库中的种子数据。

# 使用构造函数的实体类型

2018/5/3 • 7 min to read • [Edit Online](#)

## 注意

此功能是在 EF 核心 2.1 的新增功能。

从开始 EF 核心 2.1, 它则现在可以定义参数的构造函数, 并让 EF 核心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, 或到各种类型的服务, 以便于行为喜欢延迟加载。

## 注意

截至 EF 核心 2.1, 按照约定将为所有构造函数绑定。若要使用的特定构造函数的配置被计划未来的版本。

## 绑定到映射的属性

典型的博客/Post 模型, 请考虑:

```
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 核心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非参数构造函数, 然后设置每个属性的值从数据库。但是, 如果 EF 核心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; }
}

```

需要注意的一些事项：

- 需要具有构造函数的参数不是所有属性。例如，由任何构造函数参数，因此 EF 核心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匹配的属性类型和名称，只不过属性可以是 Pascal 大小写形式时参数采用 camel 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 核心使用构造函数。
- 构造函数可以是公共、私有的或具有任何其他可访问性。

### 只读属性

通过构造函数中设置属性之后它很有意义，以使其中一些只读的。EF 核心支持此功能，但有一些需要注意的事项：

- 按照约定未映射而无需 setter 的属性。（这样倾向于映射不应将映射，如计算属性的属性。）
- 使用自动生成的密钥值需要是可读写，因为密钥的值需要插入新实体时，密钥生成器进行设置的密钥属性。

若要避免这些内容的简单方法是使用专用 setter。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 核心看到为读写模式，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 setter 的替代方法是使属性实际上是只读的并在 OnModelCreating 中添加更多显式映射。同样，可以完全删除某些属性并将其替换为只有字段。例如，考虑这些实体类型：

```

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; }
}

```

和中 OnModelCreating 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);
        });
}

```

需要注意的事项：

- "Property"的密钥现在是一个字段。它不是 `readonly` 字段，以便可以使用由存储生成的键。
- 其他属性是只读属性只能在构造函数中设置。
- 如果主键值是永远只有由 EF 设置，或从数据库读取，则无需将其包含在构造函数。这为简单字段离开键"属性"，并使其清除，它不应设置显式创建新的博客或文章时。

### 注意

此代码将导致编译器警告"169", 该字段永远不会使用该值。由于在现实中 EF 核心 extralinguistic 的方式使用该字段, 这可予以忽视。

## 将注入服务

EF 核心还可以将"服务"注入到实体类型的构造函数。例如, 以下可插入:

- `DbContext` -还为您派生的 `DbContext` 类型类型化的当前上下文实例
- `ILazyLoader` 的延迟加载服务-请参阅[延迟加载文档](#)有关详细信息
- `Action<object, string>` 的延迟加载委托-请参阅[延迟加载文档](#)有关详细信息
- `IEntityType` -与此实体类型关联的 EF 核心元数据

### 注意

截至 EF 核心 2.1, 可插入仅通过 EF 核心已知的服务。支持将注入应用程序服务正在考虑针对将来的版本。

例如, 插入的 `DbContext` 可用来有选择地访问数据库以获取有关相关的实体的信息不加载所有这些情况下。在以下示例中这用于不加载文章情况下获取的博客中的文章数:

```
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; }
}
```

有关此请注意以下内容:

- 构造函数是专用容器, 因为它只能由 EF 核心, 并且没有用于常规用途的另一个公共构造函数。

- 使用插入的服务的代码（即上下文）是对其防御性正在 `null` 以便处理在 EF 核心不创建实例的情况。
- 由于服务存储在读/写属性，则将重置时该实体附加到新的上下文实例。

#### 警告

因为它将直接与 EF 核心的实体类型，将注入如下 DbContext 通常被视为反模式。使用如下服务注入之前请仔细考虑所有选项。



# 拥有的实体类型

2018/4/23 • 6 min to read • [Edit Online](#)

## 注意

此功能是在 EF 核心 2.0 中的新增功能。

EF 核心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。这些应用程序称为\_拥有实体类型\_。包含拥有的实体类型的实体是其\_所有者\_。

## 显式配置

拥有永远不会包含类型由 EF 核心模型中按照约定的实体。你可以使用 `OwnsOne` 中的方法 `OnModelCreating` 或批注与类型 `OwnedAttribute` (新在 EF 核心 2.1) 配置为拥有类型的类型。

在此示例中, `StreetAddress` 是没有标识属性的类型。它用作 `Order` 类型的属性来指定特定订单的发货地址。在 `OnModelCreating`, 我们使用 `OwnsOne` 方法, 以指定 `ShippingAddress` 属性是拥有顺序类型的实体。

```
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; }
}

// OnModelCreating
modelBuilder.Entity<Order>().OwnsOne(p => p.ShippingAddress);
```

如果该 `ShippingAddress` 属性是在顺序类型中为私有, 则可以使用的字符串版本 `OwnsOne` 方法:

```
modelBuilder.Entity<Order>().OwnsOne(typeof(StreetAddress), "ShippingAddress");
```

在此示例中, 我们使用 `OwnedAttribute` 来实现相同的目的:

```
[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; }
}
```

## 隐式键

在 EF 核心 2.0 和 2.1 中, 仅引用导航属性可以指向拥有的类型。不支持拥有类型的集合。这些引用拥有类型始终与所有者具有一对一的关系, 因此它们不需要其自己的密钥值。在前面的示例中, 不需要定义键属性 `StreetAddress` 类型。

在了解如何 EF 核心跟踪这些对象的顺序, 有必要考虑主密钥被创建为[隐藏属性](#)拥有的类型。拥有类型的实例键的值将作为所有者实例的密钥的值相同。

## 映射拥有与表拆分的类型

在使用关系数据库时, 按照约定拥有类型映射到与所有者相同的表。这要求拆分成两个表: 某些列将用于存储的所有者, 数据和某些列将用于存储数据的拥有的实体。这是一个称作表拆分的常见功能。

### 提示

拥有表拆分与存储的类型可以是使用非常类似于复杂类型中使用 `ef6` 更高版本。

按照约定, EF 核心将遵循模式在拥有的实体类型的属性的名称的数据库列 `_EntityType_OwnedEntityType_`。因此 `StreetAddress` 属性将显示在具有名 `ShippingAddress_Street` 和 `ShippingAddress_City` `Orders` 表中。

你可以将附加 `HasColumnName` 方法来重命名这些列。在其中 `StreetAddress` 是公共属性的情况下, 映射将是

```
modelBuilder.Entity<Order>().OwnsOne(
    o => o.ShippingAddress,
    sa =>
    {
        sa.Property(p=>p.Street).HasColumnName("ShipsToStreet");
        sa.Property(p=>p.City).HasColumnName("ShipsToCity");
    });
```

## 共享多个类型中拥有相同的.NET 类型

拥有的实体类型可以是相同的.NET 类型作为另一个拥有的实体类型, 因此.NET 类型可能不够来标识拥有的类型。

在这些情况下, 从所有者指向拥有的实体的属性将成为`_定义导航_拥有的实体类型`。从 EF 核心的角度来看, 定义导航是标识的旁边的.NET 类型的类型的一部分。

例如, 在以下类中, `ShippingAddress` 和 `BillingAddress` 都相同的.NET 类型, `StreetAddress`:

```
public class Order
{
    public int Id { get; set; }
    public StreetAddress ShippingAddress { get; set; }
    public StreetAddress BillingAddress { get; set; }
}
```

若要了解如何 EF 核心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。

## 嵌套拥有的类型

在此示例中 `OrderDetails` 拥有 `BillingAddress` 和 `ShippingAddress`, 它们是两个 `StreetAddress` 类型。然后 `OrderDetails` 归顺序类型。

```

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

```

可以[链接到](#) `OwnsOne` fluent 映射以配置此模型中的方法：

```

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

```

它是可能实现相同的操作使用 `OwnedAttribute` `OrderDetails` 和 `StreetAddress` 上。

嵌套拥有除类型之外，拥有的类型可以引用常规实体。在下面的示例中，国家/地区是常规的（即非自有）实体：

```

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

```

此功能设置以从 EF6 脱离复杂类型拥有的实体类型。

## 存储拥有单独的表中的类型

也拥有的类型从 EF6 复杂与类型不同，它可以存储在所有者从单独的表。若要重写将拥有的类型映射到与所有者相同的表的约定，你可以轻松调用 `.ToTable` 并提供不同的表名。下面的示例将 `OrderDetails` 和其两个地址映射到单独的表中的顺序：

```

modelBuilder.Entity<Order>().OwnsOne(p => p.OrderDetails, od =>
{
    od.OwnsOne(c => c.BillingAddress);
    od.OwnsOne(c => c.ShippingAddress);
}).ToTable("OrderDetails");

```

## 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，即使拥有的类型存储在单独的表。根据之前所述的模型，以下查询将拉取顺序、`OrderDetails` 和从数据库的所有挂起订单两个拥有的 `StreetAddresses`：

```
var orders = context.Orders.Where(o => o.Status == OrderStatus.Pending);
```

## 限制

以下是拥有的实体类型的一些限制。这些限制一些基本到如何拥有类型的工作，但某些其他属性我们想要删除在将来版本的时间点限制：

### 当前不足之处

- 不支持的所有类型的继承
- 不能由集合导航属性指向拥有的类型
- 因为它们使用表分割默认拥有类型还具有以下限制，除非显式映射到另一个表：
  - 它们不能拥有由派生类型
  - 定义导航属性不能设置为 null（即拥有对同一个表的类型并总是必需的）

### 按设计限制

- 无法创建 `DbSet<T>`
- 不能调用 `Entity<T>()` 拥有类型 `ModelBuilder`

# 查询类型

2018/5/15 • 3 min to read • [Edit Online](#)

## 注意

此功能是 EF 核心 2.1 中的新增功能

EF 核心模型可以包含实体类型，除了\_查询类型\_，这可以用于执行针对未映射到实体类型的数据的数据库查询。

查询类型的实体类型具有许多相似之处：

- 它们也可以添加到模型或者在 `OnModelCreating`，或通过"set"上的属性，派生 `DbContext`。
- 它们支持许多相同的映射功能，如继承映射，导航属性（请参阅下面的限制），然后在关系存储，能够配置的目标数据库对象和通过 fluent API 方法或数据注释的列。

但是值不同于实体中的类型，它们：

- 不需要一个密钥，以定义。
- 永远不会跟踪的更改上 `DbContext` 并因此永远不会插入、更新或删除数据库上。
- 永远不会发现的约定。
- 仅支持一部分导航映射功能-具体而言：
  - 它们可能永远不会充当一种关系的主体端。
  - 它们只能包含指向实体的引用导航属性。
  - 实体不能包含为查询类型的导航属性。
- 得到解决 `ModelBuilder` 使用 `Query` 方法而不是 `Entity` 方法。
- 在映射 `DbContext` 类型的属性通过 `DbQuery<T>` 而不是 `DbSet<T>`
- 映射到使用的数据库对象 `ToListAsync` 方法，而不是 `ToListAsync`。
- 可以将映射到\_定义查询\_定义查询是在充当查询类型的数据源的模型中声明的辅助查询。

下面是一些的查询类型主要使用方案：

- 即席充当的返回类型 `FromSql()` 查询。
- 映射到数据库视图。
- 将映射到没有定义的主键的表。
- 将映射到模型中定义的查询。

## 提示

将查询类型映射到的数据库对象使用实现 `ToListAsync` fluent API。此方法中指定的数据库对象是从 EF 核心的角度来看，视图，这意味着它将被视为只读查询源和不能作为目标的更新、插入或删除操作。但是，这并不意味着该数据库对象是实际需要的数据库视图-还可以是以只读方式将被视为数据库表。相反，实体类型，EF 核心假定数据库对象中指定 `ToListAsync` 方法可以视为\_表\_、意味着它可以用作查询源，但是还针对由更新、删除和插入操作。事实上，可以指定的名称中的数据库视图 `ToListAsync` 和所有内容应该可以满足要求，只要视图配置为在数据库上可更新。

## 示例

下面的示例演示如何使用查询类型来查询数据库视图。

## 提示

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 Post 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, 使我们能够查询数与每个博客文章:

```
db.Database.ExecuteSqlCommand(
    @"CREATE VIEW View_BlogPostCounts AS
    SELECT Name, Count(p.PostId) as PostCount from Blogs b
    JOIN Posts p on p.BlogId = b.BlogId
    GROUP BY b.Name");
```

接下来, 我们定义一个类以保留的数据库视图的结果:

```
public class BlogPostsCount
{
    public string BlogName { get; set; }
    public int PostCount { get; set; }
}
```

接下来, 我们配置中的查询类型\_OnModelCreating\_使用 `modelBuilder.Query<T>` API。我们可以使用标准 fluent 配置 Api 来配置查询类型的映射:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder
        .Query<BlogPostsCount>().ToView("View_BlogPostCounts")
        .Property(v => v.BlogName).HasColumnName("Name");
}
```

最后, 我们可以查询数据库视图中的标准方式:

```
var postCounts = db.BlogPostCounts.ToList();

foreach (var postCount in postCounts)
{
    Console.WriteLine($"{postCount.BlogName} has {postCount.PostCount} posts.");
    Console.WriteLine();
}
```

#### 提示

请注意我们还定义了上下文级别查询属性 (DbQuery) 使其作为针对此类型的查询的根。

# 使用相同的 DbContext 类型的多个模型之间交替

2018/3/1 • 1 min to read • [Edit Online](#)

生成的模型 `OnModelCreating` 无法使用属性在上下文更改生成模型的方式。例如它无法用于排除某些属性：

```
public class DynamicContext : DbContext
{
    public bool? IgnoreIntProperty { get; set; }

    public DbSet<ConfigurableEntity> Entities { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        => optionsBuilder
            .UseInMemoryDatabase("DynamicContext")
            .ReplaceService<IModelCacheKeyFactory, DynamicModelCacheKeyFactory>();

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        if (IgnoreIntProperty.HasValue)
        {
            if (IgnoreIntProperty.Value)
            {
                modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.IntProperty);
            }
            else
            {
                modelBuilder.Entity<ConfigurableEntity>().Ignore(e => e.StringProperty);
            }
        }
    }
}
```

## IModelCacheKeyFactory

但是如果你尝试执行上述无需进行其他更改则会得到相同的模型每次创建新的上下文时的任何值的

`IgnoreIntProperty`。这引起缓存 EF 使用通过仅调用来提高性能的机制的模型 `OnModelCreating` 一次，并且缓存模型。

默认情况下 EF 假定，对于任何给定的上下文类型模型处于相同。若要实现此目的的默认实现

`IModelCacheKeyFactory` 返回上下文类型中只包含一个键。若要更改此你需要更换 `IModelCacheKeyFactory` 服务。需要对使用其他模型项返回可比较的对象的新实现 `Equals` 考虑会影响模型的所有变量的方法：

```
public class DynamicModelCacheKeyFactory : IModelCacheKeyFactory
{
    public object Create(DbContext context)
    {
        if (context is DynamicContext dynamicContext)
        {
            return (context.GetType(), dynamicContext.IgnoreIntProperty);
        }
        return context.GetType();
    }
}
```



# 关系数据库建模

2018/1/23 • 1 min to read • [Edit Online](#)

一般而言，本部分中的配置适用于关系数据库。安装关系数据库提供程序时，此处显示的扩展方法将变为可用（原因在于共享的 `Microsoft.EntityFrameworkCore.Relational` 包）。

# 表映射

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

表映射标识应从查询的表数据, 并将其保存到数据库中。

## 约定

按照约定, 每个实体将安装程序以将映射到与同名表 `DbSet<TEntity>` 公开派生上下文上的该实体的属性。如果没有 `DbSet<TEntity>` 包含对于给定的实体中, 使用的类名称。

## 数据注释

数据注释可用于配置类型可以映射到表。

```
using System.ComponentModel.DataAnnotations.Schema;
```

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

你还可以指定此表所属的架构。

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

## Fluent API

Fluent API 可用于配置类型可以映射到表。

```
using Microsoft.EntityFrameworkCore;
```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .ToTable("blogs");
    }
}

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

你还可以指定此表所属的架构。

```
modelBuilder.Entity<Blog>()
    .ToTable("blogs", schema: "blogging");
```

# 列映射

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

应从查询哪些列数据, 并将其保存到数据库中, 将标识列映射。

## 约定

按照约定, 每个属性将是安装程序以映射到与属性同名的列。

## 数据注释

数据注释可用于配置的属性映射的列。

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

## Fluent API

Fluent API 可用于配置的属性映射的列。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.BlogId)
            .HasColumnName("blog_id");
    }
}

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

# 数据类型

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

数据类型是指数据库特定类型的属性映射的列。

## 约定

按照约定, 数据库提供程序选择基于属性的 CLR 类型的数据类型。它还会考虑其他元数据, 例如配置[最大长度](#), 无论该属性是一部分的主键, 等等。

例如, SQL Server 使用 `datetime2(7)` 为 `DateTime` 属性, 和 `nvarchar(max)` 为 `string` 属性 (或 `nvarchar(450)` 为 `string` 用作键的属性)。

## 数据注释

可以使用数据注释以指定确切的数据类型列。

例如下面的代码配置 `url` 为非 unicode 字符串, 最大长度为 `200` 和 `Rating` 用作小数点的精度 `5` 和缩放的 `2`。

```
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; }
}
```

## Fluent API

Fluent API 还可用于指定列的相同数据类型。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>(eb =>
        {
            eb.Property(b => b.Url).HasColumnType("varchar(200)");
            eb.Property(b => b.Rating).HasColumnType("decimal(5, 2)");
        });
    }
}

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

# 主键

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

每个实体类型的键是引入了主键约束。

## 约定

按照约定, 将名为数据库中的主键 `PK_<type name>`。

## 数据注释

可以使用数据注释不配置为主键的任何关系数据库特定方面。

## Fluent API

Fluent API 可用于配置数据库中的主键约束的名称。

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

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

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

# 默认架构

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

默认架构是将在中创建对象, 如果为该对象未显式配置架构的数据库架构。

## 约定

按照约定, 数据库提供程序将选择最合适的默认架构。例如, Microsoft SQL Server 将使用 `dbo` 架构和 SQLite 将不使用架构 (因为 SQLite 中不支持架构)。

## 数据注释

不能设置使用数据注释的默认架构。

## Fluent API

可以使用 Fluent API 指定默认架构。

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

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



# 计算列

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

计算的列是在数据库中计算其值的列。计算的列可以使用表中其他列, 若要计算其值。

## 约定

按照约定, 计算的列不在模型中创建。

## 数据注释

计算的列不可以使用数据注释配置。

## Fluent API

可以使用 Fluent API 指定属性应映射到计算列。

```
class MyContext : DbContext
{
    public DbSet<Person> People { get; set; }

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

public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string DisplayName { get; set; }
}
```

# 序列

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

序列生成顺序的数字值在数据库中。序列不与特定表相关联。

## 约定

按照约定, 序列不会引入中对模型。

## 数据注释

你可以配置使用数据注释的序列。

## Fluent API

Fluent API 可用于在模型中创建一个序列。

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers");
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
    public string Url { get; set; }
}
```

你还可以配置序列, 如其架构、起始值和增量的其他的方面。

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
            .StartsAt(1000)
            .IncrementsBy(5);
    }
}
```

后一个序列中引入, 可用于在您的模型中生成属性的值。例如, 你可以使用[默认值](#)从序列中插入的下一步的值。

```
class MyContext : DbContext
{
    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")
            .StartsAt(1000)
            .IncrementsBy(5);

        modelBuilder.Entity<Order>()
            .Property(o => o.OrderNo)
            .HasDefaultValueSql("NEXT VALUE FOR shared.OrderNumbers");
    }
}

public class Order
{
    public int OrderId { get; set; }
    public int OrderNo { get; set; }
    public string Url { get; set; }
}
```

# 默认值

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

列的默认值是如果插入新行, 但未指定值的列将插入的值。

## 约定

按照约定, 未配置默认值。

## 数据注释

不可以设置使用数据注释的默认值。

## Fluent API

可以使用 Fluent API 指定属性的默认值。

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

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

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

你还可以指定用于计算默认值的 SQL 片段。

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

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

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

# 索引

2018/3/1 • 1 min to read • [Edit Online](#)

## 注意

一般而言, 本部分中的配置适用于关系数据库。安装关系数据库提供程序时, 此处显示的扩展方法将变为可用(原因在于共享的 Microsoft.EntityFrameworkCore.Relational 包)。

关系数据库中的索引映射到索引中的实体框架的核心概念相同。

## 约定

按照约定, 名为索引 `IX_<type name>_<property name>`。对于组合索引 `<property name>` 将成为下划线分隔列表的属性名称。

## 数据注释

不能使用数据注释配置索引。

## Fluent API

Fluent API 可用于配置索引的名称。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .HasName("Index_Url");
    }
}

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

你还可以指定筛选器。

```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .HasFilter("[Url] IS NOT NULL");
    }
}

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

```

当使用 SQL Server 提供程序 EF 将添加 IS NOT NULL 筛选所有可以为 null 的列都是唯一索引的一部分。若要重写你可以提供此约定 `null` 值。

```

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasIndex(b => b.Url)
            .IsUnique()
            .HasFilter(null);
    }
}

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

```

# 外键约束

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下, 此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序 (由于共享 *Microsoft.EntityFrameworkCore.Relational* 包)。

外键约束是为每个关系在模型中引入的。

## 约定

按照约定, 名为外键约束 `FK_<dependent type name>_<principal type name>_<foreign key property name>`。复合外键 `<foreign key property name>` 将成为外键属性名称下划线分隔列表。

## 数据注释

外键约束名称不能使用数据注释配置。

## Fluent API

可以使用 Fluent API 来配置关系的外键约束名称。

```
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.BlogId)
            .HasConstraintName("ForeignKey_Post_Blog");
    }
}

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; }
}
```



# 备用键（唯一约束）

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

一般情况下，此部分中的配置是适用于关系数据库。此处所示的扩展方法将变为可用时安装关系数据库提供程序（由于共享 *Microsoft.EntityFrameworkCore.Relational* 包）。

唯一约束是每个备用键在模型中引入的。

## 约定

索引和约束引入了备用密钥相关的命名约定，由 `AK_<type name>_<property name>`。有关备用组合键 `<property name>` 将成为下划线分隔列表的属性名称。

## 数据注释

不能使用数据注释配置唯一约束。

## Fluent API

可以使用 Fluent API 来配置备用项的索引和约束的名称。

```
class MyContext : DbContext
{
    public DbSet<Car> Cars { get; set; }

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

class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
}
```

# 继承（关系数据库）

2018/2/13 • 2 min to read • [Edit Online](#)

## 注意

一般而言, 本部分中的配置适用于关系数据库。安装关系数据库提供程序时, 此处显示的扩展方法将变为可用(原因在于共享的 Microsoft.EntityFrameworkCore.Relational 包)。

EF 模型中的继承用于控制如何在数据库中表示的实体类中的继承。

## 注意

目前, 每个层次结构一个表的 (TPH) 模式在 EF 核心实现。其他常见模式如每种类型一个表 (TPT) 和表的每种具体的类型 (TPC) 尚不可用。

## 约定

按照约定, 将使用的每个层次结构一张表 (TPH) 模式映射继承。TPH 使用单个表来存储层次结构中的所有类型的数据。鉴别器列用于标识的每一行代表的类型。

在模型中显式包含两个或多个继承的类型时, EF 核心将仅安装程序继承 (请参阅[继承](#)有关详细信息)。

下面是一个示例, 演示一个简单的继承方案和使用 TPH 模式的关系数据库表中存储的数据。鉴别器列标识哪种类型的博客存储在每行。

```
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; }
}
```

Results				
	BlogId	Discriminator	Url	RssUrl
1	1	Blog	http://blogs.msdn.com/dotnet	NULL
2	2	RssBlog	http://blogs.msdn.com/adonet	http://blogs.msdn.com/b/adonet/atom.aspx

## 数据注释

数据注释不能用于配置继承。

## Fluent API

Fluent API 可用于配置的名称和类型鉴别器列和用于标识层次结构中的每种类型的值。

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasDiscriminator<string>("blog_type")
            .HasValue<Blog>("blog_base")
            .HasValue<RssBlog>("blog_rss");
    }
}

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

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

## 配置鉴别器属性

在上面的示例中，鉴别器创建作为[隐藏属性](#)的基实体的层次结构。由于这是在模型中的属性，可以将它配置其他属性一样。例如，若要在使用默认值，通过约定鉴别器时设置的最大长度：

```
modelBuilder.Entity<Blog>()
    .Property("Discriminator")
    .HasMaxLength(200);
```

鉴别器还可以映射到你的实体中的实际 CLR 属性中。例如：

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .HasDiscriminator<string>("BlogType");
    }
}

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

public class RssBlog : Blog
{
    public string RssUrl { get; set; }
}
```

组合这两项操作就可以同时将鉴别器映射到某个实际属性并将其配置：

```
modelBuilder.Entity<Blog>(b =>
{
    b.HasDiscriminator<string>("BlogType");

    b.Property(e => e.BlogType)
        .HasMaxLength(200)
        .HasColumnName("blog_type");
});
```

# 查询数据

2018/5/25 • 1 min to read • [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 .NET 语言) 基于派生上下文和实体类编写强类型查询。LINQ 查询的一种表示形式会传递给数据库提供程序, 进而转换为特定于数据库的查询语言(例如, 适用于关系数据库的 SQL)。有关如何处理查询的更多详细信息, 请参阅[查询的工作原理](#)。

# 基本查询

2018/5/31 • 1 min to read • [Edit Online](#)

了解如何使用语言集成查询 (LINQ) 从数据库加载实体。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 101 个 LINQ 示例

此页面显示了多个使用 Entity Framework Core 完成常见任务的示例。有关显示 LINQ 可完成的任务的大量示例, 请参阅 [101 个 LINQ 示例](#)。

## 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);
}
```

## 筛选

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Where(b => b.Url.Contains("dotnet"))
        .ToList();
}
```

# 加载相关数据

2018/5/31 • 9 min to read • [Edit Online](#)

Entity Framework Core 允许你在模型中使用导航属性来加载相关实体。有三种常见的 O/RM 模式可用于加载相关数据。

- “预先加载”表示从数据库中加载相关数据，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相关数据。
- “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相关数据。

## 提示

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相关数据。在以下示例中，结果中返回的博客将使用相关文章填充其 `Posts` 属性。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ToList();
}
```

## 提示

Entity Framework Core 会将导航属性自动修复为之前加载到上下文实例中的任何其他实体。因此，即使不显式包含导航属性的数据，但如果先前加载了部分或所有相关实体，则仍可能填充该属性。

可以将来自多个关系的相关数据包含在单个查询中。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Include(blog => blog.Owner)
        .ToList();
}
```

## 包括多个级别

可以深化到关系以使用 `ThenInclude` 方法包括相关数据的多个级别。以下示例加载了所有博客、其相关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();
}
```

### 注意

当前版本的 Visual Studio 提供的代码完成选项不正确, 因此在集合导航属性之后使用 `ThenInclude` 方法时, 可能会导致正确的表达式标记有语法错误。这是在 <https://github.com/dotnet/roslyn/issues/8237> 中跟踪的 IntelliSense bug 的症状。只要代码正确无误且能够成功编译, 就可以放心地忽略这些虚假语法错误。

可以用链锁住对 `ThenInclude` 的多个调用以继续包括相关数据的更深级别。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .ThenInclude(post => post.Author)
        .ThenInclude(author => author.Photo)
        .ToList();
}
```

可以全部合并以将来自多个级别和多个根的相关数据包含在同一查询中。

```
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();
}
```

可能希望包括所包含的实体之一的多个相关实体。例如, 当查询 `Blog` 时, 将包括 `Posts`, 然后希望同时包括 `Posts` 的 `Author` 和 `Tags`。为此, 需要在根目录开头指定每个包含路径。例如, `Blog -> Posts -> Author` 和 `Blog -> Posts -> Tags`。这并不意味着你将获得冗余联接, 在大多数情况下, 当生成 SQL 时, EF 将合并联接。

```
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();
}
```

### 在派生类型上包括

可以使用 `Include` 和 `ThenInclude` 包括来自仅在派生类型上定义的导航的相关数据。

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; }
}

```

所有人员(可以使用许多模式预先加载的学生)的 `School` 导航的内容:

- 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()
```

- 使用采用类型 `string` 的参数的 `Include` 的重载

```
context.People.Include("Student").ToList()
```

## 忽略包含

如果更改查询, 从而使其不再返回查询以之为开头的实体类型的实例, 则会忽略 `include` 运算符。

在以下示例中, `include` 运算符基于 `Blog`, 但 `Select` 运算符用于更改查询以返回匿名类型。在这种情况下, `include` 运算符没有任何效果。

```
using (var context = new BloggingContext())
{
    var blogs = context.Blogs
        .Include(blog => blog.Posts)
        .Select(blog => new
        {
            Id = blog.BlogId,
            Url = blog.Url
        })
        .ToList();
}
```

默认情况下, 当忽略 include 运算符时, EF Core 将记录警告。有关查看日志记录输出的详细信息, 请参阅[日志记录](#)。当忽略 include 运算符以引发异常或不执行任何操作时, 可以更改行为。这是在为上下文(如果使用的是 ASP.NET Core, 则通常在 `DbContext.OnConfiguring` 或 `Startup.cs` 中)设置选项时完成的。

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(@"Server=
(localdb)\mssqllocaldb;Database=EFQuerying;Trusted_Connection=True;ConnectRetryCount=0")
        .ConfigureWarnings(warnings => warnings.Throw(CoreEventId.IncludeIgnoredWarning));
}
```

## 显式加载

### 注意

EF Core 1.1 中已引入此功能。

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相关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 将自动设置新加载实体的导航属性以引用已加载的任何实体, 并设置已加载实体的导航属性以引用新加载的实体。

### 查询相关实体

还可以获得表示导航属性内容的 LINQ 查询。

这样可以执行诸如通过相关实体运行聚合运算符而无需将其加载到内存中等操作。

```
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();
}
```

还可以筛选将哪些相关实体加载到内存中。

```
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

### 注意

EF Core 2.1 中已引入此功能。

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);
```

或者, 使用 `AddDbContext` 时:

```
.AddDbContext<BloggingContext>(
    b => b.UseLazyLoadingProxies()
        .UseSqlServer(myConnectionString));
```

EF Core 将为可被重写的任何导航属性(即, 必须是 `virtual` 且在可从中继承的类上)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; }
}
```

### 不使用代理进行延迟加载

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;
    }
}

```

这不需要从中继承的实体类型或要虚拟化的导航属性，并允许使用 `new` 创建的实体实例在附加到上下文后进行延迟加载。但需要对 `ILazyLoader` 服务的引用，从而将实体类型结合到 EF Core 程序集。为了避免这种情况，EF Core 允许注入 `ILazyLoader.Load` 方法作为委托。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;
    }
}
```

### 注意

延迟加载委托的构造函数参数必须称为“lazyLoader”。为未来版本计划使用不同名称的配置。

## 相关数据和序列化

由于 EF Core 将自动修复导航属性，因此可以在对象图中以循环结束。例如，加载博客及其相关文章将生成引用文章集合的博客对象。其中每篇文章将返回引用该博客。

某些序列化框架不允许使用此类循环。例如，如果遇到循环，Json.NET 将引发以下异常。

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。这是在 `Startup.cs` 中通过 `ConfigureServices(...)` 方法完成的。

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddMvc()
        .AddJsonOptions(
            options => options.SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore
        );

    ...
}
```

# 客户端与服务器评估

2018/5/31 • 1 min to read • [Edit Online](#)

Entity Framework Core 支持在客户端上评估查询的各个部分，并将查询的各个部分推送到数据库。由数据库提供程序确定将在数据库中评估查询的哪些部分。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 客户端评估

在下面的示例中，Helper 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 提供程序对此方法的实现方式没有任何见解，因此不可以将其转换为 SQL。在数据库中评估查询的所有其他方面，但会在客户端上通过此方法传递返回的 `URL`。

```
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;
}
```

## 禁用客户端评估

虽然客户端评估非常有用，但在某些情况下可能会导致性能不佳。请考虑以下查询，其中 Helper 方法现已在筛选器中使用。由于无法在数据库中执行此操作，因此所有数据将被拉入内存中，然后会在客户端上应用筛选器。根据数据量以及筛选出的数据量，这可能会导致性能不佳。

```
var blogs = context.Blogs
    .Where(blog => StandardizeUrl(blog.Url).Contains("dotnet"))
    .ToList();
```

默认情况下，当执行客户端评估时，EF Core 将记录警告。有关查看日志记录输出的详细信息，请参阅[日志记录](#)。当客户端评估引发异常或不执行任何操作时，可以更改行为。这是在为上下文（如果使用的是 ASP.NET Core，则通常在 `DbContext.OnConfiguring` 或 `Startup.cs` 中）设置选项时完成的。



```
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

2018/5/31 • 2 min to read • [Edit Online](#)

跟踪行为可控制 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。Entity Framework Core 还会修复从跟踪查询中获取的实体与先前已加载到 DbContext 实例中的实体之间的导航属性。

## 提示

可在 [GitHub](#) 上查看此文章的示例。

## 跟踪查询

默认情况下，跟踪返回实体类型的查询。这表示可以对这些实体实例进行更改，并且通过 `SaveChanges()` 永久保存这些更改。

在以下示例中，将检测到对博客分级所做的更改，并在 `SaveChanges()` 期间将这些更改永久保存到数据库中。

```
using (var context = new BloggingContext())
{
    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。

可以交换单个非跟踪查询：

```
using (var context = new BloggingContext())
{
    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：

```
using (var context = new BloggingContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var blogs = context.Blogs.ToList();
}
```

### 注意

非跟踪查询仍在执行查询中执行标识解析。如果结果集多次包含相同的实体，则每次会在结果集中返回实体类的相同实例。但是，弱引用用于跟踪已返回的实体。如果具有相同标识的上一个结果超出范围，并运行垃圾回收，则可能会获得新的实体实例。有关详细信息，请参阅[查询的工作原理](#)。

## 跟踪和投影

即使查询的结果类型不是实体类型，但如果结果包含实体类型，则默认情况下也会跟踪这些实体类型。在以下返回匿名类型的查询中，会跟踪结果集中 `Blog` 的实例。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Blog = b,
                Posts = b.Posts.Count()
            });
}
```

如果结果集不包含任何实体类型，则不会执行跟踪。在以下返回匿名类型（具有实体中的某些值，但没有实际实体类型的实例）的查询中，不会执行跟踪。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs
        .Select(b =>
            new
            {
                Id = b.BlogId,
                Url = b.Url
            });
}
```

# 原生 SQL 查询

2018/5/31 • 4 min to read • [Edit Online](#)

通过 Entity Framework Core 可以在使用关系数据库时下拉到原生 SQL 查询。这在无法使用 LINQ 表示要执行的查询或在使用 LINQ 查询导致 SQL 发送到数据库的效率低时非常有用。

## 提示

可在 [GitHub](#) 上查看此文章的示例。

## 限制

使用原生 SQL 查询时需注意以下几个限制：

- SQL 查询只能用于返回属于你的模型的实体类型。积压工作有一个增强功能，即可从原生 SQL 查询中返回临时类型。
- SQL 查询必须返回实体或查询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称匹配。请注意，这与 EF6 不同，其中已忽略原生 SQL 查询的属性/列映射，且结果集列名称必须与属性名称匹配。
- SQL 查询不能包含相关数据。但是，在许多情况下可以使用 `Include` 运算符在查询顶部进行编写以返回相关数据(请参阅[包括相关数据](#))。
- 传递到此方法的 `SELECT` 语句通常应该可以进行编写：如果 EF Core 需要计算服务器上的其他查询运算符(例如，转换 `FromSql` 后应用的 LINQ 运算符)，则提供的 SQL 将被视为子查询。这意味着传递的 SQL 不应包含子查询上无效的任何字符或选项，如：
  - 尾随分号
  - 在 SQL Server 上，尾随的查询级提示，如 `OPTION (HASH JOIN)`
  - 在 SQL Server 上，不附带 `SELECT` 子句中的 `TOP 100 PERCENT` 的 `ORDER BY` 子句
- 除 `SELECT` 以外的其他 SQL 语句自动识别为不可编写。因此，存储过程的完整结果将始终返回到客户端，且在内存中计算 `FromSql` 后应用的任何 LINQ 运算符。

## 基本原生 SQL 查询

可以使用 `FromSql` 扩展方法基于原生 SQL 查询开始 LINQ 查询。

```
var blogs = context.Blogs
    .FromSql("SELECT * FROM dbo.Blogs")
    .ToList();
```

原生 SQL 查询可用于执行存储过程。

```
var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogs")
    .ToList();
```

## 传递参数

正如接受 SQL 的任何 API 一样，务必参数化任何用户输入以抵御 SQL 注入攻击。可以将参数占位符包含在 SQL 查询字符串中，然后提供参数值作为其他参数。提供的任何参数值将自动转换为 `DbParameter`。

下面的示例将一个参数传递到存储过程。尽管这看上去可能像 `String.Format` 语法，但提供的值包装在参数中，且生成的参数名称插入在指定 `{0}` 占位符的位置。

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser {0}", user)
    .ToList();
```

这是相同的查询，但使用 EF Core 2.0 及更高版本支持的字符串内插语法：

```
var user = "johndoe";

var blogs = context.Blogs
    .FromSql($"EXECUTE dbo.GetMostPopularBlogsForUser {user}")
    .ToList();
```

还可以构造 `DbParameter` 并将其作为参数值提供。这样可以在 SQL 查询字符串中使用命名参数

```
var user = new SqlParameter("user", "johndoe");

var blogs = context.Blogs
    .FromSql("EXECUTE dbo.GetMostPopularBlogsForUser @user", user)
    .ToList();
```

## 使用 LINQ 编写

如果在数据库中可以在 SQL 查询上进行编写，则可以使用 LINQ 运算符在初始原生 SQL 查询顶部进行编写。可以使用 `SELECT` 关键字在其上进行编写的 SQL 查询。

下面的示例使用可从表值函数 (TVF) 中选择原生 SQL 查询，然后使用 LINQ 在其上进行编写以执行筛选和排序。

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Where(b => b.Rating > 3)
    .OrderByDescending(b => b.Rating)
    .ToList();
```

### 包括相关数据

使用 LINQ 运算符编写可用于将相关数据包含在查询中。

```
var searchTerm = ".NET";

var blogs = context.Blogs
    .FromSql($"SELECT * FROM dbo.SearchBlogs({searchTerm})")
    .Include(b => b.Posts)
    .ToList();
```

## 警告

**始终为原生 SQL 查询使用参数化：**接受原生 SQL 字符串(如 `FromSql` 和 `ExecuteSqlCommand`)的 API 允许将值作为参数轻松传递。除了验证用户输入以外,始终为原生 SQL 查询/命令中使用的所有值使用参数化。如果使用字符串串联来动态生成查询字符串的任何部分,则你将负责验证所有输入以抵御 SQL 注入攻击。

# 异步查询

2018/5/31 • 1 min to read • [Edit Online](#)

当在数据库中执行查询时，异步查询可避免阻止线程。这有助于避免冻结客户端应用程序的 UI。异步操作还可以增加 Web 应用程序中的吞吐量，其中可以释放线程以在数据库操作完成时处理其他请求。有关详细信息，请参阅[使用 C# 异步编程](#)。

## 警告

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 提供了一组异步扩展方法，可用作导致执行查询并返回结果的 LINQ 方法的替代方法。示例包括 `ToListAsync()`、`ToArrayAsync()`、`SingleAsync()` 等。没有 LINQ 运算符(如 `Where(...)`、`OrderBy(...)` 等)的异步版本，因为这些方法仅生成 LINQ 表达式树，且不会导致在数据库中执行查询。

## 重要事项

在 `Microsoft.EntityFrameworkCore` 命名空间中定义 EF Core 异步扩展方法。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();
    }
}
```

# 查询的工作原理

2018/5/31 • 2 min to read • [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 .NET 语言) 基于派生上下文和实体类编写强类型查询。

## 查询的寿命

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. 数据库提供程序确定可以在数据库中评估查询的哪些部分
  - b. 查询的这些部分会转换为数据库特定的查询语言(例如, 关系数据库的 SQL)
  - c. 一个或多个查询会发送到数据库和返回的结果集(结果是数据库中的值, 而不是实体实例)
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已在上下文实例的更改跟踪器中的实体
    - 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. 如果这是非跟踪查询, EF 会检查数据是否表示已在此查询的结果集中的实体
    - 如果是, 则会返回现有实体<sup>(1)</sup>
    - 如果不是, 则会创建新实体并返回该新实体

<sup>(1)</sup> 非跟踪查询使用弱引用跟踪已返回的实体。如果具有相同标识的上一个结果超出范围, 并运行垃圾回收, 则可能会获得新的实体实例。

## 执行查询时

调用 LINQ 运算符时, 只会生成查询的内存中表示形式。使用结果时, 查询只会发送到数据库。

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`、`ToArray`、`Single`、`Count` 等运算符
- 将查询结果数据绑定到 UI

### 警告

**始终验证用户输入:** 当 EF 抵御 SQL 注入攻击时, 不会执行输入的任何常规验证。因此, 如果传递到 API、用于 LINQ 查询、分配给实体属性等的值来自不受信任的源, 则应按照每个应用程序要求执行相应的验证。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, 则也会需要确保只能构造预期表达式。



# 全局查询筛选器

2018/5/31 • 2 min to read • [Edit Online](#)

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 - 实体类型定义“IsDeleted”属性。
- 多租户 - 实体类型定义“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。

### 提示

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; }
}
```

请记住\_博客\_实体上 *\_tenantId* 字段的声明。这会用于将每个博客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”。即实例标记为已删除, 而实际上不会删除基础数据。

接下来, 使用 `HasQueryFilter` API 在 *OnModelCreating* 中配置查询筛选器。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().Property<string>("TenantId").HasField("_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。

#### 提示

请注意 DbContext 实例级别字段的使用: `_tenantId` 用于设置当前租户。模型级别筛选器将使用来自正确上下文实例的值。即执行查询的实例。

## 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：

- 筛选器不能包含对导航属性的引用。
- 仅可为继承层次结构的根实体类型定义筛选器。

# 保存数据

2018/5/25 • 1 min to read • [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

2018/5/31 • 2 min to read • [Edit Online](#)

了解如何使用上下文和实体类添加、修改和删除数据。

## 提示

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://sample.com" };
    context.Blogs.Add(blog);
    context.SaveChanges();

    Console.WriteLine(blog.BlogId + ": " + blog.Url);
}
```

## 提示

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。

只需修改分配给属性的值，然后调用 `SaveChanges`。

```
using (var context = new BloggingContext())
{
    var blog = context.Blogs.First();
    blog.Url = "http://sample.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删除且不再插入。

```
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。

### 注意

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成功或失败，决不部分应用这些操作。

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

using (var context = new BloggingContext())
{
    // add
    context.Blogs.Add(new Blog { Url = "http://sample.com/blog_one" });
    context.Blogs.Add(new Blog { Url = "http://sample.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

2018/5/31 • 2 min to read • [Edit Online](#)

除了独立实体以外，还可以使用模型中定义的关系。

## 提示

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，则将其中的一个添加到上下文时也会添加其他实体。

在下面的示例中，博客和三个相关文章会全部插入到数据库中。找到并添加这些文章，因为它们可以通过

`Blog.Posts` 导航属性访问。

```
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();
}
```

## 提示

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 `blog` 实体的 `Posts` 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 (`post`) 跟踪的实体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，将在 `Blog` 和 `Post` 之间的关系上配置级联删除，以便从数据库中删除 `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

2018/5/31 • 15 min to read • [Edit Online](#)

级联删除通常在数据库术语中使用，用于描述允许删除某行以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的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

### 注意

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。

对于上面的第二个操作，如果外键不可以为 null，则将某个外键值设置为 null 无效。(不可为 null 的外键相当于必选关系。)在这些情况下，EF Core 会跟踪到外键属性已标记为 null，直到调用 *SaveChanges*，由于无法将更改永久保存到数据库，因此会在此时引发异常。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。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：

行为名称	对内存中的依赖项/子项的影响	对数据库中的依赖项/子项的影响
<b>Cascade</b>	删除实体	删除实体
<b>ClientSetNull</b> (默认)	外键属性设置为 null	无
<b>SetNull</b>	外键属性设置为 null	外键属性设置为 null
<b>Restrict</b>	无	无

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：

行为名称	对内存中的依赖项/子项的影响	对数据库中的依赖项/子项的影响
<b>Cascade</b> (默认)	删除实体	删除实体
<b>ClientSetNull</b>	引发 <i>SaveChanges</i>	无



行为名称	对内存中的依赖项/子项的影响	对数据库中的依赖项/子项的影响
<b>SetNull</b>	引发 SaveChanges	引发 SaveChanges
<b>Restrict</b>	无	无

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，数据库必须支持该值，配置此类数据库可能会导致其他限制，这实际上通常使得此选项不切实际。这是“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。

**注意**

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。

**注意**

**EF Core 2.0 中的更改：**在以前版本中，“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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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, 因此它保留非 null, SaveChanges 引发异常且不保存

## 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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 '1' 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”指示 EF 不要自动将外键设置为 null, 因此它保留非 null, SaveChanges 引发异常且不保存

## 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, 将为你设置此级联行为。

# 处理并发冲突

2018/5/31 • 4 min to read • [Edit Online](#)

## 注意

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信息，请参阅[并发令牌](#)。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

**数据库并发**指的是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**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影响，因此能够成功。在最坏的情况下，两个或更多进程将尝试进行冲突更改，其中只有一个进程应该成功。

## 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 包括对任何 `UPDATE` 或 `DELETE` 语句的 `WHERE` 子句中的并发令牌值的检查。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 `Person` 上的 `LastName` 配置为并发令牌。则针对用户的任何更新操作将包括 `WHERE` 子句中的并发检查：

```
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合并。要合并的值将根据应用程序的不同而有所不同，并且可能由用户输入指示。

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 `SaveChanges` 期间捕获 `DbUpdateConcurrencyException`。
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，`Person.FirstName` 和 `Person.LastName` 设置为并发令牌。在包括特定于应用程序的逻辑以选择要保存的值的位罝中有 `// 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.ExecuteSqlCommand(
        "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

2018/5/31 • 6 min to read • [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。

## 提示

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，则会在事务中应用对 `SaveChanges()` 的单一调用中的所有更改。如果所有更改均失败，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()` 可保证完全成功，或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。如果应用程序要求被视为有必要，则应该仅手动控制事务。

## 控制事务

可以使用 `DbContext.Database` API 开始、提交和回滚事务。以下示例显示了两个 `SaveChanges()` 操作以及正在单个事务中执行的 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)

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，因为该提供程序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。

#### 提示

`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 `DbContext.OnConfiguring` 中使用的 `DbConnection`。

```
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<BloggngContext>()
    .UseSqlServer(new SqlConnection(connectionString))
    .Options;

using (var context1 = new BloggngContext(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 BloggngContext(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。

```
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<BloggngContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggngContext(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

### 注意

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 }))
{
    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<BloggngContext>()
            .UseSqlServer(connection)
            .Options;

        using (var context = new BloggngContext(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.EnlistTransaction(transaction);
            context.Database.OpenConnection();

            // 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();
        }

        // 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。尝试使用该功能时, SqlClient for .NET Core 2.0 将引发异常。

### 重要事项

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

2018/5/31 • 1 min to read • [Edit Online](#)

当所做的更改写入数据库时，异步保存可避免阻止线程。这有助于避免冻结客户端应用程序的 UI。异步操作还可以增加 Web 应用程序中的吞吐量，其中可以释放线程以在数据库操作完成时处理其他请求。有关详细信息，请参阅[使用 C# 异步编程](#)。

## 警告

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

2018/5/31 • 8 min to read • [Edit Online](#)

DbContext 实例将自动跟踪从数据库返回的实体。调用 SaveChanges 时，将检测到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，如 Web 应用程序，其中实体被查询、发送到客户端、被修改、发送回请求中的服务器，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的(应插入)还是现有的(应更新)。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## 提示

EF Core 只能跟踪具有给定主键值的任何实体的一个实例。避免使其成为问题的最佳方法是为每个工作单元使用短生存期的上下文，以便上下文一开始是空的、向其附加实体、保存这些实体，然后释放并丢弃上下文。

## 标识新实体

### 客户端标识新实体

要处理的最简单情况是客户端通知服务器实体是新的还是现有的。例如，通常插入新实体的请求与更新现有实体的请求不同。

本部分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用于确定是需要插入实体还是需要更新实体。如果未设置键(即 CLR 默认值仍为 null、零等)，则实体必须为新的，且需要插入。另一方面，如果已设置键值，则必须之前已保存该实体，且现在需要更新。换言之，如果键具有值，则实体被查询、发送到客户端、现在返回进行更新。

实体类型为未知时，可以轻松检查未设置的键：

```
public static bool IsItNew(Blog blog)
    => blog.BlogId == 0;
```

但是，EF 还具有一种内置方法可用于任何实体类型和键类型：

```
public static bool IsItNew(DbContext context, object entity)
    => !context.Entry(entity).IsKeySet;
```

## 提示

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形并确定对每个实体做什么，如使用 TrackGraph API 时。在执行任何调用以跟踪实体\_之前\_，应仅按照此处显示的方式使用键值。

### 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;
```

这超出了本文档的范围，无法显示用于从客户端传递标志的完整代码。在 Web 应用中，通常表示为不同操作发出不同请求，或传递请求中的某些状态，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。

### 提示

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 调用会将实体标记为根据需要进行更新。

#### 提示

SetValues 仅标记为已将具有不同值的属性修改为跟踪的实体中的属性。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，则完全不会发送任何更新。）

## 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析。

### 所有新实体/所有现有实体

使用图形的一个示例是插入或更新博客及其相关文章集合。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，可以使用查询和一些处理：

```
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

由于通常缺少实体表示应删除该实体，因此可能很难处理删除。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图形差异真正的内容。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

2018/5/31 • 4 min to read • [Edit Online](#)

生成的属性是在添加和/或更新实体时生成其值(由 EF 或数据库生成)的属性。有关详细信息,请参阅[生成的属性](#)。

可能会出现希望设置已生成属性的显式值,而不是生成显式值的情况。

## 提示

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,因此仍将设置为 `DateTime` 的 CLR 默认值。
- 对于第二个员工,已设置 `1-Jan-2000` 的显式值。

```
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
```

## 显式值到 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 `SaveChanges()` 之前手动启用 `IDENTITY_INSERT`。

### 注意

积压工作中有[功能请求](#), 用来在 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.ExecuteSqlCommand("SET IDENTITY_INSERT dbo.Employees ON");
        context.SaveChanges();
        context.Database.ExecuteSqlCommand("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.AfterSaveBehavior = PropertySaveBehavior.Ignore;
```

### 注意

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)。

### 注意

**EF Core 2.0 中的更改:** 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 `UPDATE` 操作期间为 `LastPayRaise` 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
```

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, 已在一周前设置显式值(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

2018/3/6 • 2 min to read • [Edit Online](#)

我们希望 EF Core 可用于任何可编写 .NET 代码 的平台, 我们仍在为实现这一目标而努力。尽管自动测试证明 EF Core 支持 .NET Core 和 .NET Framework , 同时许多应用程序已成功使用 EF Core, 但 Mono、Xamarin 和 UWP 仍存在一些问題。

下表提供了每个 .NET 实现的指南:

.NET 实现	状态	EF CORE 1.X 要求	EF CORE 2.X 要求 <sup>(1)</sup>
<b>.NET Core</b> ( <a href="#">ASP.NET Core</a> 、 <a href="#">控制台</a> 等)	完全支持且推荐使用	<a href="#">.NET Core SDK 1.x</a>	<a href="#">.NET Core SDK 2.x</a>
.NET Framework (WinForms、WPF、ASP.NET、 <a href="#">控制台</a> 等)	完全支持且推荐使用。EF6 也适用 <sup>(2)</sup>	.NET Framework 4.5.1	.NET Framework 4.6.1
<b>Mono 和 Xamarin</b>	正在进行 <sup>(3)</sup>	Mono 4.6 Xamarin.iOS 10 Xamarin.Mac 3 Xamarin.Android 7	Mono 5.4 Xamarin.iOS 10.14 Xamarin.Mac 3.8 Xamarin.Android 7.5
<a href="#">通用 Windows 平台</a>	推荐 EF Core 2.0.1 <sup>(4)</sup>	<a href="#">.NET Core UWP 5.x 包</a>	<a href="#">.NET Core UWP 6.x 包</a>

<sup>(1)</sup> EF Core 2.0 面向 .NET 实现, 因此需要支持 [.NET Standard 2.0](#) 的 .NET 实现。

<sup>(2)</sup> 请参阅[比较 EF Core 和 EF6](#) 以选择合适的技术。

<sup>(3)</sup> 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 2.0 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。

<sup>(4)</sup> 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。

对于未按预期工作的任意组合, 我们建议在 [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

2018/6/9 • 5 min to read • [Edit Online](#)

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

### 重要事项

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。

NUGET 程序包	支持的数据库引擎	维护商/供应商	备注/要求	有用的链接
<a href="#">Microsoft.EntityFrameworkCore.SqlServer</a>	SQL Server 2008 及以上版本	<a href="#">EF Core 项目</a> (Microsoft)		<a href="#">docs</a>
<a href="#">Microsoft.EntityFrameworkCore.Sqlite</a>	SQLite 3.7 及以上版本	<a href="#">EF Core 项目</a> (Microsoft)		<a href="#">docs</a>
<a href="#">Microsoft.EntityFrameworkCore.InMemory</a>	EF Core 内存中数据库	<a href="#">EF Core 项目</a> (Microsoft)	仅用于测试	<a href="#">docs</a>
<a href="#">Npgsql.EntityFrameworkCore.PostgreSQL</a>	postgresql	<a href="#">Npgsql 开发团队</a>		<a href="#">docs</a>
<a href="#">Pomelo.EntityFrameworkCore.MySql</a>	MySQL、MariaDB	<a href="#">Pomelo Foundation 项目</a>		<a href="#">自述文件</a>
<a href="#">Pomelo.EntityFrameworkCore.MyCat</a>	MyCAT 服务器	<a href="#">Pomelo Foundation 项目</a>	预发行版，最新为 EF Core 1.1	<a href="#">自述文件</a>
<a href="#">EntityFrameworkCore.SqlServerCompact40</a>	SQL Server Compact 4.0	<a href="#">Erik Ejlskov Jensen</a>	.NET Framework	<a href="#">wiki</a>
<a href="#">EntityFrameworkCore.SqlServerCompact35</a>	SQL Server Compact 3.5	<a href="#">Erik Ejlskov Jensen</a>	.NET Framework	<a href="#">wiki</a>
<a href="#">MySql.Data.EntityFrameworkCore</a>	MySQL	<a href="#">MySQL 项目</a> (Oracle)	预发行版	<a href="#">docs</a>
<a href="#">FirebirdSql.EntityFrameworkCore.Firebird</a>	Firebird 2.5 和 3.x	<a href="#">Jiří Činčura</a>	EF Core 2.0 及以上版本	<a href="#">docs</a>
<a href="#">EntityFrameworkCore.FirebirdSql</a>	Firebird 2.5 和 3.x	<a href="#">Rafael Almeida</a>	EF Core 2.0 及以上版本	<a href="#">wiki</a>

NUGET 程序包	支持的数据库引擎	维护商/供应商	备注/要求	有用的链接
<a href="#">IBM.EntityFrameworkCore</a>	Db2、Informix	<a href="#">IBM</a>	Windows 版本	<a href="#">博客</a>
<a href="#">IBM.EntityFrameworkCore-Inx</a>	Db2、Informix	<a href="#">IBM</a>	Linux 版本	<a href="#">博客</a>
<a href="#">IBM.EntityFrameworkCore-osx</a>	Db2、Informix	<a href="#">IBM</a>	macOS 版本	<a href="#">博客</a>
<a href="#">Devart.Data.Oracle.EFCore</a>	Oracle 9.2.0.4 及以上版本	<a href="#">DevArt</a>	已付	<a href="#">docs</a>
<a href="#">Devart.Data.PostgreSql.EFCore</a>	PostgreSQL 8.0 及以上版本	<a href="#">DevArt</a>	已付	<a href="#">docs</a>
<a href="#">Devart.Data.SQLite.EFCore</a>	SQLite 3 及以上版本	<a href="#">DevArt</a>	已付	<a href="#">docs</a>
<a href="#">Devart.Data.MySql.EFCore</a>	MySQL 5 及以上版本	<a href="#">DevArt</a>	已付	<a href="#">docs</a>
<a href="#">EntityFrameworkCore.Jet</a>	Microsoft Access 文件	<a href="#">Bubi</a>	EF Core 2.0, .NET Framework	<a href="#">自述文件</a>

## 未来的提供程序

### Cosmos DB

我们一直在为 Cosmos DB 中的 DocumentDB API 开发 EF Core 提供程序。这将是我们推出的首个完整的文档型数据库提供程序，此次尝试得来的经验将用于改进 2.1 之后版本的设计。目前我们计划在 2.1 时间框架中发布提供程序的早期预览版。

### Oracle

Oracle .NET 团队表示他们计划在 2018 年第三季度左右针对 EF Core 2.0 发布第一方提供程序。有关详细信息，请参阅其[关于 .NET Core 和 Entity Framework Core 方向的声明](#)。请在 [Oracle 社区网站](#)上提出有关此提供程序的任何问题(包括发布时间线)。

与此同时，EF 团队还为[Oracle 数据库提供了示例 EF Core 提供程序](#)。推行该项目不是为了生产属于 Microsoft 的 EF Core 提供程序，而是为了帮助我们找出 EF Core 的关系和基础功能间的差距，我们需要弥补这些差距，更好地为 Oracle 提供支持，加快 Oracle 或第三方为 EF Core 开发其他 Oracle 提供程序的速度。

对实现该示例有益的建议我们都会考虑。我们也欢迎并鼓励社区以该示例为基础为 EF Core 创建开源 Oracle 提供程序。

## 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。这意味着可使用命令行中的 `dotnet` 工具来安装它们：

```
dotnet add package provider_package_name
```

或者在 Visual Studio 中，使用 NuGet 包管理器控制台：

```
install-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。

#### 重要事项

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

2018/3/6 • 1 min to read • [Edit Online](#)

此数据库提供程序允许将 Entity Framework Core 与 Microsoft SQL Server (包括 SQL Azure) 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。

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

## 开始操作

下列资源可帮助你开始使用此提供程序。

- [在 .NET Framework \(控制台、WinForms、WPF 等\) 上开始使用](#)
- [在 ASP.NET Core 上开始使用](#)
- [UnicornStore 示例应用程序](#)

## 支持的数据库引擎

- Microsoft SQL Server (2008 及以上版本)

## 支持的平台

- .NET Framework (4.5.1 及以上版本)
- .NET 核心
- Mono (4.2.0 及以上版本)

Caution: Using this provider on Mono will make use of the Mono SQL Client implementation, which has a number of known issues. For example, it does not support secure connections (SSL).

# 内存优化表支持在 SQL Server EF 核心数据库提供程序

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

EF 核心 1.1 中已引入此功能。

**内存优化表**是整个表驻留在内存中的 SQL 服务器的功能。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。

## 配置的内存优化表

你可以指定一个实体映射到表是内存优化。当使用 EF 核心创建和维护的数据库基于你的模型（通过迁移或 `Database.EnsureCreated()`），将为这些实体创建内存优化表。

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ForSqlServerIsMemoryOptimized();
}
```

# SQLite EF Core 数据库提供程序

2018/3/6 • 1 min to read • [Edit Online](#)

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。

```
Install-Package Microsoft.EntityFrameworkCore.Sqlite
```

## 开始操作

下列资源可帮助你开始使用此提供程序。

- [UWP 上的本地 SQLite](#)
- [.NET Core 应用程序到新 SQLite 数据库](#)
- [Unicorn Clicker 示例应用程序](#)
- [Unicorn Packer 示例应用程序](#)

## 支持的数据库引擎

- SQLite(3.7 及以上版本)

## 支持的平台

- .NET Framework(4.5.1 及以上版本)
- .NET 核心
- Mono(4.2.0 及以上版本)
- 通用 Windows 平台

## 限制

有关 SQLite 提供程序的一些重要限制, 请参阅 [SQLite 限制](#)。

# SQLite EF 核心数据库提供程序限制

2018/3/3 • 1 min to read • [Edit Online](#)

SQLite 提供程序具有许多的迁移限制。这些限制的大多数是基础的 SQLite 数据库引擎中的限制的结果，并不特定于 EF。

## 建模限制

(实体框架关系数据库提供程序通过共享) 的公共关系库定义建模概念所共有的大多数关系数据库引擎的 Api。SQLite 提供程序不支持几个这些概念。

- 架构
- 序列

## 迁移限制

SQLite 数据库引擎不支持大量的大部分其他关系数据库支持的架构操作。如果你尝试将不受支持的操作之一应用于一个 SQLite 数据库则 `NotSupportedException` 将引发。

操作	支持？	需要版本
AddColumn	✓	1.0
AddForeignKey	✗	
AddPrimaryKey	✗	
AddUniqueConstraint	✗	
AlterColumn	✗	
CreateIndex	✓	1.0
CreateTable	✓	1.0
DropColumn	✗	
DropForeignKey	✗	
DropIndex	✓	1.0
DropPrimaryKey	✗	
DropTable	✓	1.0
DropUniqueConstraint	✗	
RenameColumn	✗	

操作	支持？	需要版本
RenameIndex	✓	2.1
RenameTable	✓	1.0
EnsureSchema	✓（无操作）	2.0
DropSchema	✓（无操作）	2.0
Insert	✓	2.0
更新	✓	2.0
删除	✓	2.0

## 迁移限制解决方法

一些可以解决方法通过手动编写代码，在你迁移执行表中这些限制的重新生成。表重新生成涉及重命名现有表、创建新表、将数据复制到新的表中，和删除旧表。你将需要使用 `Sql(string)` 方法来执行这些步骤中的一部分。

请参阅[进行其他类型的表架构更改](#)有关更多详细信息的 SQLite 文档中。

将来，EF 可能支持某些操作通过使用下的表重新生成方法。你可以[此功能跟踪我们的 GitHub 项目](#)。

# EF Core In-Memory 数据库提供程序

2018/3/6 • 1 min to read • [Edit Online](#)

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。

```
Install-Package Microsoft.EntityFrameworkCore.InMemory
```

## 开始操作

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

- 内置的内存中数据库（仅用于测试目的）

## 支持的平台

- .NET Framework (4.5.1 及以上版本)
- .NET 核心
- Mono (4.2.0 及以上版本)
- 通用 Windows 平台

# 编写数据库提供程序

2018/3/1 • 1 min to read • [Edit Online](#)

有关编写实体框架核心数据库提供程序的信息，请参阅[因此你想要编写 EF 核心提供程序](#)通过Arthur Vickers。

基本 EF 核心代码属于开放源代码，包含多个数据库提供程序可以使用作为参考。你可以在<https://github.com/aspnet/EntityFrameworkCore> 找到源代码。

## 提供程序注意标签

一旦开始于提供程序的工作，监视 `providers-beware` 在我们的 GitHub 问题和拉取请求的标签。我们使用此标签来标识可能会影响提供程序编写器的更改。

## 建议的第三方提供程序命名

我们建议使用 NuGet 包的以下命名。这是与 EF 核心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`

# 管理数据库架构

2018/1/23 • 1 min to read • [Edit Online](#)

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。若要二者择一，请确定真实源是 EF Core 模型还是数据库架构。

如果希望 EF Core 模型为真实源，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。

如果希望数据库架构为真实源，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 模型来建立 DbContext 和实体类型类的基架。

## 注意

[创建和删除 API](#) 也可从 EF Core 模型创建数据库架构。但是，它们主要用于测试、原型制作以及可接受删除数据库的其他方案。



# 迁移

2018/6/11 • 4 min to read • [Edit Online](#)

通过迁移能够以递增方式将架构更改应用到数据库，使其与 EF Core 模型保持同步，同时保留数据库中的现有数据。

## 创建数据库

定义初始模型后，即应创建数据库。为此，需要添加初始迁移。请安装 [EF Core 工具](#) 并运行相应命令。

```
Add-Migration InitialCreate
```

```
dotnet ef migrations add InitialCreate
```

向“迁移”目录下的项目添加以下三个文件：

- **00000000000000000000\_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)。
- **00000000000000000000\_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。

### 提示

可以自由移动“迁移”文件并更改其命名空间。创建的新迁移属于上个迁移的同级。

接下来，将迁移应用到数据库以创建架构。

```
Update-Database
```

```
dotnet ef database update
```

## 添加另一个迁移

更改 EF Core 模型后，数据库架构将不同步。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，如果我作出更改并保存客户对产品的评论，则可以选择类似于 `AddProductReviews` 的内容。

```
Add-Migration AddProductReviews
```

```
dotnet ef migrations add AddProductReviews
```

搭建迁移基架后，应检查迁移的准确性，并添加正确应用迁移所需的任何其他操作。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。为了进行改善, 请按如下所示进行重写。

```
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");
```

#### 提示

如果在基架化某一操作时添加新的迁移, 系统会提醒你此举可能导致数据丢失(例如删除某列)。请务必仔细检查这些迁移的准确性。

使用相应命令将迁移应用到数据库。

```
Update-Database
```

```
dotnet ef database update
```

## 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。

```
Remove-Migration
```

```
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)，但需要将其复原，则可使用应用迁移的相同命令并指定要回退的迁移名称。

```
Update-Database LastGoodMigration
```

```
dotnet ef database update LastGoodMigration
```

## 空迁移

有时添加迁移而不进行任何模型更改很有用处。在这种情况下，添加新迁移会创建一个空迁移。可以自定义此迁移，执行与 EF Core 模型不直接相关的操作。可能希望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
- 等等

## 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。

```
Script-Migration
```

```
dotnet ef migrations script
```

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。

请注意，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。

```
myDbContext.Database.Migrate();
```

#### 警告

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。

#### 注意

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 `DbContext.GetService<IMigrator>()` 进行访问。

# 在团队环境中的迁移

2018/1/23 • 1 min to read • [Edit Online](#)

在团队环境中使用迁移，请额外注意模型快照文件。此文件会告诉您是否队友的迁移会合并完全与您的是否您需要通过重新创建你迁移共享它之前解决冲突。

## 合并

从你的团队成员合并迁移，则可能会模型快照文件中显示冲突。如果这两个更改是不相关，合并为普通和两个迁移可以共存。例如，你会收到类似如下所示的客户实体类型配置中的合并冲突：

```
<<<<<< Mine
b.Property<bool>("Deactivated");
=====
b.Property<int>("LoyaltyPoints");
>>>>>> Theirs
```

由于这两种属性需要在最终模型中存在，请通过添加这两个属性完成合并。在许多情况下，版本控制系统可能自动为你合并此类更改。

```
b.Property<bool>("Deactivated");
b.Property<int>("LoyaltyPoints");
```

在这些情况下，你的迁移和队友的迁移是相互独立。由于其中任一个都可能首先得到应用，你不必对你的迁移之前与团队共享进行任何其他更改。

## 解决冲突

有时你会遇到 true 冲突时合并模型快照模型。例如，您和队友可能每个已重命名相同的属性。

```
<<<<<< Mine
b.Property<string>("Username");
=====
b.Property<string>("Alias");
>>>>>> Theirs
```

如果遇到这种类型的冲突，请重新创建迁移以解决它。请执行这些步骤：

1. 中止的合并和回滚到在合并前工作目录
2. 删除你的迁移（但保留模型更改）
3. 将团队成员那里的更改合并到你的工作目录
4. 重新添加你的迁移

执行此操作后，两个迁移可以应用正确的顺序。其迁移已重命名的列的第一次，应用 *别名*，此后您迁移将其重命名为 *用户名*。

你的迁移可以安全地共享与团队的其余部分。

# 自定义迁移操作

2018/5/15 • 2 min to read • [Edit Online](#)

MigrationBuilder API 允许你迁移期间，执行许多不同类型的操作，但它并不详尽。但是，API 还会可扩展它允许你定义自己的操作。有两种方法来扩展 API：使用 `Sql()` 方法，或通过定义自定义 `MigrationOperation` 对象。

为了说明，让我们看一下实现创建使用每种方法的数据库用户的操作。在我们迁移中，我们想要启用编写以下代码：

```
migrationBuilder.CreateUser("SQLUser1", "Password");
```

## 使用 MigrationBuilder.Sql()

实现自定义操作的最简单方法是定义扩展方法的调用 `MigrationBuilder.Sql()`。下面是生成相应的 Transact SQL 示例。

```
static MigrationBuilder CreateUser(  
    this MigrationBuilder migrationBuilder,  
    string name,  
    string password)  
=> migrationBuilder.Sql($"CREATE USER {name} WITH PASSWORD '{password}';");
```

如果你迁移需要支持多个数据库提供程序，则可以使用 `MigrationBuilder.ActiveProvider` 属性。下面是支持 Microsoft SQL Server 和 PostgreSQL 一个示例。

```
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}';");  
    }  
}
```

此方法仅适用如果知道每个提供程序将应用你自定义操作的位置。

## 使用 MigrationOperation

若要分离 SQL 的自定义操作，你可以定义自己 `MigrationOperation` 来表示它。然后，因此它可以确定相应的 SQL 生成，系统会操作传递给提供程序。

```
class CreateUserOperation : MigrationOperation
{
    public string Name { get; set; }
    public string Password { get; set; }
}
```

使用此方法，该扩展方法只需添加到这些操作之一 `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商联系，以了解如何在此操作生成 SQL 其 `IMigrationsSqlGenerator` 服务。下面是示例重写以处理新的操作的 SQL Server 的生成器。

```

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.TypeMapper.GetMapping(typeof(string));

        builder
            .Append("CREATE USER ")
            .Append(sqlHelper.DelimitIdentifier(name))
            .Append(" WITH PASSWORD = ")
            .Append(stringMapping.GenerateSqlLiteral(password))
            .AppendLine(sqlHelper.StatementTerminator)
            .EndCommand();
    }
}

```

将替换为更新默认迁移 sql 生成器服务。

```

protected override void OnConfiguring(DbContextOptionsBuilder options)
=> options
    .UseSqlServer(connectionString)
    .ReplaceService<IMigrationsSqlGenerator, MyMigrationsSqlGenerator>();

```



# 使用一个单独的项目

2018/1/23 • 1 min to read • [Edit Online](#)

你可能想要比另一个包含其他程序集中存储迁移你 `DbContext`。此外可以使用此策略，维护多个组迁移，例如，一个用于开发，另一个版本的升级。

若要执行此操作...

1. 创建一个新的类库。
2. 添加到 `DbContext` 程序集的引用。
3. 将迁移和模型快照文件移动到类库。
  - 如果你尚未添加任何，添加一个到 `DbContext` 项目，然后将其移。
4. 配置迁移程序集：

```
options.UseSqlServer(  
    connectionString,  
    x => x.MigrationsAssembly("MyApp.Migrations"));
```

5. 添加到迁移程序集中的启动程序集的引用。
  - 如果这导致循环依赖关系，更新类库的输出路径：

```
<PropertyGroup>  
    <OutputPath>..\MyStartupProject\bin\$(Configuration)\</OutputPath>  
</PropertyGroup>
```

如果你这样做的所有内容正确，你应能够向项目添加新的迁移。

```
Add-Migration NewMigration -Project MyApp.Migrations
```

```
dotnet ef migrations add NewMigration --project MyApp.Migrations
```

# 多个提供程序的迁移

2018/3/21 • 1 min to read • [Edit Online](#)

**EF 核心工具**仅创建为活动提供程序的迁移的基架。有时，但是，你可能想要用于多个提供程序（例如 Microsoft SQL Server 和 SQLite）的 DbContext。有两种方法来处理这与迁移。你可以维护两组迁移-的情况下，为每个提供程序--或合并到单个设置的其中一个可以工作。

## 两个迁移集

在第一个方法中，你将生成两个迁移的每个模型更改。

一种办法这是将每个迁移集中**单独的**程序集和手动添加两个迁移之间进行切换的活动提供程序（和迁移程序集）。

轻松使用这些工具的另一种方法是创建从 DbContext 派生，并重写活动提供程序的新类型。此类型用于在设计时添加或将应用迁移的时间。

```
class MySqliteDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=my.db");
}
```

### 注意

由于每个迁移集使用其自己的 DbContext 类型，此方法不需要使用具有单独迁移的程序集。

在添加新的迁移时，指定上下文类型。

```
Add-Migration InitialCreate -Context MyDbContext -OutputDir Migrations\SqlServerMigrations
Add-Migration InitialCreate -Context MySqliteDbContext -OutputDir Migrations\SqliteMigrations
```

```
dotnet ef migrations add InitialCreate --context MyDbContext --output-dir Migrations/SqlServerMigrations
dotnet ef migrations add InitialCreate --context MySqliteDbContext --output-dir Migrations/SqliteMigrations
```

### 提示

你不需要指定后续迁移的输出目录，因为它们将创建为与最后一个同级。

## 有一次迁移集

如果你不喜欢具有两个集的迁移，你可以手动将它们合并到一组可以应用于两个提供程序中。

批注可以共存，因为提供程序会忽略它不了解任何批注。例如，适用于 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),
```

如果操作只能应用一个提供程序上（或它们以不同的方式提供程序之间），使用 `ActiveProvider` 属性来通知的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

2018/1/23 • 1 min to read • [Edit Online](#)

默认情况下, EF 核心将跟踪的哪些迁移已应用于数据库中名为的表记录 `__EFMigrationsHistory`。由于各种原因, 你可能想要自定义此表来更好地满足您的需要。

## 重要事项

如果你自定义迁移历史记录表后应用迁移, 你将负责更新数据库中的现有表。

## 架构和表名称

你可以更改架构和表名称使用 `MigrationsHistoryTable()` 中的方法 `OnConfiguring()` (或 `ConfigureServices()` ASP.NET Core 上)。此处是一个示例使用 SQL Server EF 核心提供程序。

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options.UseSqlServer(
        connectionString,
        x => x.MigrationsHistoryTable("__MyMigrationsHistory", "mySchema"));
```

## 其他更改

若要配置的表的其他方面, 覆盖, 并将特定于提供程序的 `IHistoryRepository` 服务。下面是更改到的 `MigrationId` 列名称的一个示例在 SQL Server 上。

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
    => options
        .UseSqlServer(connectionString)
        .ReplaceService<IHistoryRepository, MyHistoryRepository>();
```

## 警告

`SqlServerHistoryRepository` 放在内部命名空间内, 在将来版本可能会更改。

```
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

2018/1/23 • 1 min to read • [Edit Online](#)

## 注意

尚未编写此主题！ 你可以跟踪此状态 [问题](#) 通过我们公共 GitHub 问题跟踪程序。了解如何 [参与](#) GitHub 上。

# □ 反向工程的影响

2018/5/3 • 1 min to read • [Edit Online](#)

## 注意

尚未编写此主题！ 你可以跟踪此状态 [问题](#) 通过我们公共 GitHub 问题跟踪程序。了解如何 [参与](#) GitHub 上。

# Entity Framework Core 工具

2018/3/12 • 2 min to read • [Edit Online](#)

Entity Framework Core 工具有助于 EF Core 应用的开发。它们主要用于通过对数据库架构进行反向工程来建立 DbContext 和实体类型的基架，以及用于管理迁移。

EF Core 包管理器控制台 (PMC) 工具在 Visual Studio 内提供卓越体验。使用 NuGet 的包管理器控制台运行这些工具。这些工具同时适用于 .NET Framework 和 .NET Core 项目。

EF Core .NET 命令行工具是 .NET Core 命令行接口 (CLI) 工具(这些工具是跨平台的，可在 Visual Studio 之外运行)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 PMC 工具，因为这些工具提供更完整的体验。

## 框架

这些工具支持面向 .NET Framework 或 .NET Core 的项目。

如果想要使用类库，如有可能，请考虑使用 .NET Core 或 .NET Framework 类库。这样，使用 .NET 工具的问题最少。而如果想要使用 .NET Standard 类库，则需要使用面向 .NET Framework 或 .NET Core 的启动项目，使该工具具有可在其中加载类库的具体目标平台。此启动项目可以是不包含实际代码的虚拟项目 - 需要它的唯一理由是为工具提供一个目标。

如果项目面向其他框架(例如，通用 Windows 或 Xamarin)，则需要创建一个单独的 .NET Standard 类库。在这种情况下，请按照上述指南，也创建一个工具可使用的启动项目。

## 启动项目和目标项目

每当调用命令时，始终会涉及两个项目：目标项目和启动项目。

目标项目是在其中添加任何文件(或在某些情况下删除文件)的项目。

启动项目是执行项目代码时由工具模拟的项目。

目标项目和启动项目可以相同。

# EF 核心程序包管理器控制台工具

2018/5/3 • 3 min to read • [Edit Online](#)

EF 核心程序包管理器控制台 (PMC) 工具使用的 NuGet 在 Visual Studio 内运行[程序包管理器控制台](#)。这些工具同时适用于 .NET Framework 和 .NET Core 项目。

## 提示

未使用 Visual Studio？[EF 核心命令行工具](#)是跨平台，且在命令提示符下运行。

## 安装工具

通过安装 Microsoft.EntityFrameworkCore.Tools NuGet 包安装 EF 核心程序包管理器控制台 Tools。可以通过执行以下命令内的安装[程序包管理器控制台](#)。

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

如果一切运行正常，你应能够运行此命令：

```
Get-Help about_EntityFrameworkCore
```

## 提示

如果你的启动项目面向.NET 标准[跨目标支持的框架](#)之前使用的工具。

## 重要事项

如果你使用通用 **Windows**或**Xamarin**，将 EF 代码移动到标准.NET 类库和[跨目标支持的框架](#)之前使用的工具。指定为启动项目的类库。

## 使用的工具

每当调用某命令时，有两个项目涉及：

目标项目是在其中添加任何文件(或在某些情况下删除文件)的项目。目标项目默认为[默认项目](#)选择在[程序包管理器控制台](#)中，但也可以通过使用指定的项目参数。

启动项目是执行项目代码时由工具模拟的项目。它将默认为一个设为[启动项目](#)在[解决方案资源管理器](#)。它可以还指定使用-StartupProject 参数。

通用参数：

-上下文<字符串>	若要使用 DbContext。
-项目<字符串>	要使用的项目。



-StartupProject<字符串 >	要使用的启动项目。
-Verbose	显示详细输出。

若要显示有关命令的帮助信息，请使用 PowerShell 的 `Get-Help` 命令。

**提示**

上下文、Project 和 StartupProject 参数支持选项卡扩展。

**提示**

设置`env:ASPNETCORE_ENVIRONMENT`之前运行若要指定 ASP.NET Core 环境。

# 命令

## 添加迁移

将添加一个新迁移。

参数：

-名称<字符串 >	迁移的名称。
-OutputDir<字符串 >	目录（及其子命名空间）使用。路径是相对于项目目录。默认值为"迁移"。

**注意**

中的参数**粗体**是必需的以及在发生*斜体*是位置。

## 删除数据库

删除数据库。

参数：

-WhatIf	显示的数据库会被丢弃，但没有删除它。

## Get-DbContext

获取有关 DbContext 类型的信息。

## 删除迁移

删除上次的迁移。

参数：

-Force	如果它已应用到数据库，请还原迁移。

### 基架 DbContext

基架数据库类型 DbContext 和实体的类型。

参数：

<i>连接</i> <字符串 >	数据库的连接字符串。
<i>-Provider</i> <String>	要使用的提供程序。(例如, Microsoft.EntityFrameworkCore.SqlServer)
-OutputDir<字符串 >	要将文件放入的目录。路径是相对于项目目录。
-ContextDir<字符串 >	要将 DbContext 文件放入的目录。路径是相对于项目目录。
-上下文<字符串 >	若要生成的 dbcontext 名称。
-架构<String [] >	要生成实体类型的表架构。
-表<String [] >	要生成实体类型的表。
-DataAnnotations	使用属性来配置该模型（如果可能）。如果省略, 则使用仅 fluent API。
-UseDatabaseNames	使用直接从数据库表和列名称。
-Force	覆盖现有文件。

### Script-Migration

从迁移中生成的 SQL 脚本。

参数：

<i>-从</i> <字符串 >	开始迁移。默认值为 0（初始数据库）。
<i>到</i> <字符串 >	结束的迁移。默认到最后一个迁移。
幂等性	生成可以在任何迁移的数据库使用的脚本。
<i>-输出</i> <字符串 >	要将结果写入的文件。

**提示**

收件人、从, 和输出参数支持选项卡扩展。

### 更新数据库

<i>迁移</i> <字符串 >	目标迁移。如果为"0", 将恢复所有迁移。默认到最后一个迁移。

**提示**

迁移参数支持选项卡扩展。

# EF 核心.NET 命令行工具

2018/6/6 • 5 min to read • [Edit Online](#)

实体框架核心.NET 命令行工具是一种扩展到跨平台`dotnet`命令，是一部分的.NET 核心 SDK。

## 提示

如果你使用 Visual Studio, 我们建议[PMC 工具](#)相反因为它们提供了更多集成的体验。

## 安装工具

### 注意

.NET 核心 SDK 版本 2.1.300 和更高版本包括`dotnet ef`与 EF 核心 2.0 和更高版本兼容的命令。因此如果你使用的最新版本的.NET 核心 SDK 和 EF 核心运行时，需要进行任何安装，你可以忽略本部分的其余部分。

另一方面，`dotnet ef`工具包含在.NET 核心 SDK 版本 2.1.300 和更高版本不兼容与 EF Core 版本 1.0 和 1.1。你可以使用.NET 核心 sdk 2.1.300 的计算机使用 EF 核心这些早期版本的项目或更高版本安装之前，你还必须安装版本 2.1.200 或更低版本的 sdk 和配置应用程序通过修改使用该旧版本其 `global.json` 文件。此文件通常包含在解决方案目录（一个上面项目）。然后，您可以继续下面 installation 指令。

对于.NET 核心 sdk 的早期版本，你可以安装使用这些步骤的 EF 核心.NET 命令行工具：

1. 编辑项目文件并将 `Microsoft.EntityFrameworkCore.Tools.DotNet` 添加为 `DotNetCliToolReference` 项（见下文）
2. 运行以下命令：

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

生成的项目应如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
                      Version="2.0.0"
                      PrivateAssets="All" />
  </ItemGroup>
  <ItemGroup>
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
                          Version="2.0.0" />
  </ItemGroup>
</Project>
```

注意

一个具有的包引用 `PrivateAssets="All"` 意味着它不公开给引用此项目，这是非常适合通常仅在开发期间使用的包的项目。

如果您还没有完全正确，您应能够成功的命令提示中运行以下命令。

```
dotnet ef
```

## 使用的工具

每当调用某命令时，有两个项目涉及：

目标项目是在其中添加任何文件(或在某些情况下删除文件)的项目。目标项目默认为当前目录中的项目，但可以使用更改 `-项目` 选项。

启动项目是执行项目代码时由工具模拟的项目。它也默认为当前目录中的项目，但可以使用更改 `-启动项目` 选项。

注意

例如，更新的 web 应用程序具有不同的项目中安装的 EF 核心数据库将如下所示：

```
dotnet ef database update --project {project-path} (从您的 web 应用程序目录)
```

常用的选项：

	<code>--json</code>	显示 JSON 输出。
<code>-c</code>	<code>-上下文&lt;DBCONTEXT &gt;</code>	若要使用 DbContext。
<code>-p</code>	<code>-项目&lt;项目 &gt;</code>	要使用的项目。
<code>-s</code>	<code>-启动项目&lt;项目 &gt;</code>	要使用的启动项目。
	<code>-framework &lt;FRAMEWORK &gt;</code>	目标框架中。
	<code>-配置&lt;配置 &gt;</code>	要使用的配置。
	<code>-运行时&lt;标识符 &gt;</code>	若要使用运行时。
<code>-h</code>	<code>-帮助</code>	显示帮助信息。
<code>-v</code>	<code>-verbose</code>	显示详细输出。
	<code>--no-color</code>	不为着色输出。
	<code>--prefix-output</code>	输出与级别的前缀。

提示

若要指定 ASP.NET Core 环境，设置 `ASPNETCORE_ENVIRONMENT` 之前运行的环境变量。

# 命令

## dotnet ef 数据库 除去

删除数据库。

选项:

-f	--force	不确认。
	-试运行	显示的数据库会被丢弃, 但没有删除它。

## dotnet ef 数据库 更新

到指定的迁移更新数据库。

自变量:

<迁移>	目标迁移。如果为 0, 将恢复所有迁移。默认到最后一个迁移。

## dotnet ef dbcontext 信息

获取有关 DbContext 类型的信息。

## dotnet ef dbcontext 列表

列出可用的 DbContext 类型。

## dotnet ef dbcontext 基架

基架数据库类型 DbContext 和实体的类型。

自变量:

<连接>	数据库的连接字符串。
<提供程序>	要使用的提供程序。(例如 Microsoft.EntityFrameworkCore.SqlServer)

选项:

-d	-数据批注	使用属性来配置该模型(如果可能)。如果省略, 则使用仅 fluent API。
-c	-上下文<名称>	DbContext 名称。
	-上下文 dir<路径>	要将 DbContext 文件放入的目录。路径是相对于项目目录。
-f	--force	覆盖现有文件。
-o	-输出 dir<路径>	要将文件放入的目录。路径是相对于项目目录。

	-架构<SCHEMA_NAME >...	要生成实体类型的表架构。
-t	-表<TABLE_NAME >...	要生成实体类型的表。
	-使用数据库名称	使用直接从数据库表和列名称。

### dotnet ef 迁移添加

将添加一个新迁移。

自变量:

<名称 >		迁移的名称。

选项:

-o	-输出 dir<路径 >	目录（及其子命名空间）使用。路径是相对于项目目录。默认值为"迁移"。

### dotnet ef 迁移列表

列出可用的迁移。

### dotnet ef 迁移删除

删除上次的迁移。

选项:

-f	--force	如果它已应用到数据库, 请还原迁移。

### dotnet ef 迁移脚本

从迁移中生成的 SQL 脚本。

自变量:

<FROM>		开始迁移。默认值为 0（初始数据库）。
<到 >		结束的迁移。默认到最后一个迁移。

选项:

-o	-输出<文件 >	要将结果写入的文件。
-i	-幂等	生成可以在任何迁移的数据库使用的脚本。

# 设计时 DbContext 创建

2018/4/23 • 2 min to read • [Edit Online](#)

某些 EF 核心工具命令 (例如, [迁移](#) 命令) 需要派生 `DbContext` 实例以收集有关应用程序的详细信息在设计时创建实体类型和它们如何映射到数据库架构。在大多数情况下, 最好的 `DbContext` 从而创建配置如何将以类似方式在运行时配置。

有多种方法工具尝试创建 `DbContext` :

## 从应用程序服务

如果你的启动项目, ASP.NET Core 应用工具将尝试从应用程序的服务提供程序获取 `DbContext` 对象。

该工具第一次尝试通过调用获取服务提供程序 `Program.BuildWebHost()` 和访问 `IWebHost.Services` 属性。

### 注意

在创建新的 ASP.NET 核心 2.0 应用程序时, 默认情况下包含此挂钩。在以前版本的 EF Core 和 ASP.NET Core, 工具尝试调用 `Startup.ConfigureServices` 直接以获取应用程序的服务提供程序, 但此模式不能再正常 ASP.NET 核心 2.0 应用程序中。如果您正在升级到 2.0 ASP.NET Core 1.x 应用程序, 则可以修改你 `Program` 类遵循新模式。

`DbContext` 本身以及任何依赖项在其构造函数中的需要注册为在应用程序的服务提供程序的服务。这可以轻松实现通过让上的构造函数 `DbContext` 采用的实例 `DbContextOptions<TContext>` 作为自变量和使用 `AddDbContext<TContext>` 方法。

## 使用不带任何参数的构造函数

如果不能从应用程序服务提供程序获得 `DbContext`, 工具查找派生 `DbContext` 项目中的类型。然后, 它们尝试创建使用不带任何参数的构造函数实例。这可以是默认构造函数, 如果 `DbContext` 使用配置 `OnConfiguring` ] 6 方法。

## 从设计时工厂

您还可以判断工具如何通过实现来创建你 `DbContext` `IDesignTimeDbContextFactory<TContext>` 接口: 如果找到实现此接口的类中派生的同一项目 `DbContext` 或在应用程序的启动项目中, 工具绕过改为创建的 `DbContext` 和使用设计时工厂的其他方法。



```
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);
        }
    }
}
```

#### 注意

`args` 参数是当前未使用。没有[问题](#)跟踪指定的工具的设计时自变量的功能。

设计时工厂可能特别有用，如果你需要配置 `DbContext` 以不同方式为设计时比在运行时，如果 `DbContext` 构造函数采用其他参数在 DI 中，不会进行注册，如果你根本不使用 DI 或如果由于某些原因不希望让 `BuildWebHost` ASP.NET Core 应用程序中的方法

`Main` 类。

# 设计时服务

2018/1/23 • 1 min to read • [Edit Online](#)

在设计时仅使用由工具使用某些服务。这些服务 EF 核心运行时服务，以阻止它们不会与你的应用部署分开管理。若要覆盖这些服务（例如服务来生成迁移文件）之一，添加的实现 `IDesignTimeServices` 到你的启动项目。

```
class MyDesignTimeServices : IDesignTimeServices
{
    public void ConfigureDesignTimeServices(IServiceCollection services)
        => services.AddSingleton<IMigrationsCodeGenerator, MyMigrationsCodeGenerator>()
}
```

# EF Core 工具和扩展

2018/3/12 • 3 min to read • [Edit Online](#)

工具和扩展为 Entity Framework Core 提供了额外功能。

## 重要事项

扩展由多种源生成, 并不会作为 Entity Framework Core 项目的组成部分进行维护。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。

## 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。

[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。可使用 Model-First 和 Database-First 方法来设计 ORM 模型并为其生成 C# 或 Visual Basic .NET 代码。它引入了新方法设计 ORM 模型、提高工作效率并促进数据库应用程序的开发。

[网站](#)

### EF Core Power Tools

Visual Studio 2017 及以上版本扩展。可从现有数据库或 SQL Server 数据库项目反向 DbContext 和 POCO 类的工程, 并采用各种方式可视化和检查 DbContext。

[GitHub wiki](#)

## 扩展

### Microsoft.EntityFrameworkCore.AutoHistory

Microsoft.EntityFrameworkCore 的一个插件, 支持自动记录数据更改历史记录。

[GitHub 存储库](#)

### Microsoft.EntityFrameworkCore.DynamicLinq

Microsoft.EntityFrameworkCore 的 Dynamic Linq 扩展, 添加了异步支持

[GitHub 存储库](#)

### EFCore.Practices

尝试在支持测试的 API 中捕获一些良好做法或最佳做法 - 包括用于扫描 N+1 查询的小框架。

[GitHub 存储库](#)

### EFSecondLevelCache.Core

二级缓存库。二级缓存是一个查询缓存。EF 命令的结果将存储在该缓存中, 这样相同的 EF 命令将从该缓存检索其数据, 而不是再次针对数据库进行执行。

[GitHub 存储库](#)

### **Detached.EntityFramework**

加载和保存整个分离式实体图(具有其子实体和列表的实体)。灵感来自 [GraphDiff](#)。此想法也添加了一些插件来简化一些重复任务,例如审计和分页。

[GitHub 存储库](#)

### **EntityFrameworkCore.PrimaryKey**

从任何作为字典的实体中检索主键(包括复合键)。

[GitHub 存储库](#)

### **EntityFrameworkCore.Rx**

为 Entity Framework 实体的常用可观察对象重新激活扩展包装器。

[GitHub 存储库](#)

### **EntityFrameworkCore.Triggers**

使用插入、更新和删除事件将触发器添加到实体中。每个事件具有三种情况:之前、之后和失败时。

[GitHub 存储库](#)

### **EntityFrameworkCore.TypedOriginalValues**

获取对实体属性的 OriginalValue 的类型化访问权限。支持简单的和复杂的属性,不支持导航/集合。

[GitHub 存储库](#)

### **Geco**

Geco 提供一个支持复数形式/单数形式的反向模型生成器,和基于 C# 6.0 内插字符串且在 .Net Core 上运行的可编辑模板。它还提供一个具有 SQL Merge 脚本的源脚本生成器和一个脚本运行程序。

[Github 存储库](#)

### **LinqKit.Microsoft.EntityFrameworkCore**

LinqKit.Microsoft.EntityFrameworkCore 是一组适用于 LINQ to SQL 和 EntityFrameworkCore 高级用户的免费扩展。支持 Include(...) 和 IDbAsync。

[GitHub 存储库](#)

### **NeinLinq.EntityFrameworkCore**

NeinLinq.EntityFrameworkCore 提供对以下操作有所帮助的扩展:使用 LINQ 提供程序(例如仅支持一小部分 .NET 函数的 Entity Framework)、重用函数、重写查询、甚至让它们处于 null safe 状态以及使用可转换谓词和选择器生成动态查询。

[GitHub 存储库](#)

### **Microsoft.EntityFrameworkCore.UnitOfWork**

Microsoft.EntityFrameworkCore 的一个插件,支持存储库、工作模式单元和多个具有支持的分布式事务的数据库。

[GitHub 存储库](#)

### **EntityFramework.LazyLoading**

EF Core 1.1 延迟加载

[GitHub 存储库](#)

### **EFCore.BulkExtensions**

用于批量操作(插入、更新、删除)的 EntityFrameworkCore 扩展。



# 连接字符串

2018/1/23 • 2 min to read • [Edit Online](#)

大多数的数据库提供程序需要某种形式的连接字符串以连接到数据库。有时，此连接字符串包含需要保护的敏感信息。你可能还需要根据环境，如开发、测试和生产环境之间移动你的应用程序更改连接字符串。

## .NET framework 应用程序

.NET framework 应用程序，如 WinForms、WPF、控制台中，和 ASP.NET 4 中，具有的经过验证和测试连接字符串模式。应将该连接字符串添加到你的应用程序 App.config 文件 (如果你使用 ASP.NET Web.config)。如果你的连接字符串包含敏感信息，如用户名和密码，你可以保护的配置文件使用的内容[受保护的配置](#)。

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

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

### 提示

`providerName` 上 EF 核心连接字符串存储在 App.config，因为通过代码配置了数据库提供程序不需要设置。

然后可以读取连接字符串使用 `ConfigurationManager` 中上下文的 API `OnConfiguring` 方法。你可能需要添加对的引用 `System.Configuration` framework 程序集要能够使用此 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)

在 UWP 应用程序中的连接字符串通常是一个 SQLite 连接，只需指定本地中的文件名。它们通常不包含敏感信息，并不需要更改，因为应用程序部署。在这种情况下，这些连接字符串将通常可以在代码中，保留的如下所示。如果你想要使它们摆脱代码 UWP 支持设置的概念，请参阅[UWP 文档的应用设置部分](#)有关详细信息。

```
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 核心中配置系统非常灵活，并且连接字符串可存储在 `appsettings.json`，环境变量、用户机密存储中或另一个配置源。请参阅[的 ASP.NET 核心文档中的配置节](#)有关详细信息。下面的示例演示中存储的连接字符串

`appsettings.json`。

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

上下文通常在配置 `Startup.cs` 正在从配置中读取的连接字符串。请注意 `GetConnectionString()` 方法查找它的键是一个配置值 `ConnectionStrings:<connection string name>`。

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

# 日志记录

2018/6/9 • 2 min to read • [Edit Online](#)

## 提示

可在 [GitHub](#) 上查看此文章的示例。

## ASP.NET 核心应用程序

与 ASP.NET Core 的日志记录机制自动集成, EF 核心每当 `AddDbContext` 或 `AddDbContextPool` 使用。因此, 在使用 ASP.NET Core, 日志记录应配置中所述[ASP.NET 核心文档](#)。

## 其他应用程序

当前日志记录的 EF 核心要求 `ILoggerFactory` 这是配置了一个或多个 `ILoggerProvider` 本身。常见的提供程序随附于下列包:

- [Microsoft.Extensions.Logging.Console](#): 简单的控制台记录器。
- [Microsoft.Extensions.Logging.AzureAppServices](#): 支持 Azure 应用程序服务诊断日志和日志流功能。
- [Microsoft.Extensions.Logging.Debug](#): 日志传输到使用 `System.Diagnostics.Debug.WriteLine()` 调试器监视器。
- [Microsoft.Extensions.Logging.EventLog](#): 到 Windows 事件日志的日志。
- [Microsoft.Extensions.Logging.EventSource](#): 支持 `EventSource/EventListener`。
- [Microsoft.Extensions.Logging.TraceSource](#): 到使用 `System.Diagnostics.TraceSource.TraceEvent()` 的跟踪侦听器的日志。

在安装相应的程序包后, 应用程序应创建 `LoggerFactory` 的单一实例/全局实例。例如, 使用控制台记录器:

```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[] {new ConsoleLoggerProvider((_, __) => true, true)});
```

此 singleton/全局实例应然后注册与 EF 核心上 `DbContextOptionsBuilder`。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");
```

## 警告

它是重要的应用程序不会创建每个上下文实例的新 `ILoggerFactory` 实例。这样将导致内存泄漏和性能低下。

## 筛选记录的内容,

筛选所记录内容的最简单方法是在注册 `ILoggerProvider` 时对其进行配置。例如:



```
public static readonly LoggerFactory MyLoggerFactory
    = new LoggerFactory(new[]
    {
        new ConsoleLoggerProvider((category, level)
            => category == DbLoggerCategory.Database.Command.Name
                && level == LogLevel.Information, true)
    });
```

在此示例中，日志筛选以返回仅消息：

- Microsoft.EntityFrameworkCore.Database.Command 类别中
- 在信息级别

对于 EF 核心记录器类别在中定义 `DbLoggerCategory` 类，以便可以方便地查找类别，但这些解析为简单的字符串。

在基础的日志记录基础结构上的更多详细信息可在[ASP.NET 核心日志记录文档](#)。

# 连接复原

2018/1/23 • 5 min to read • [Edit Online](#)

连接复原自动重试失败的数据库命令。功能可以提供的“执行策略”，封装检测故障，然后重试命令所需的逻辑与任何数据库一起使用。EF 核心提供程序可以提供定制为其特定数据库失败条件和最佳的重试策略的执行策略。

例如，SQL Server 提供程序包括专门定制到 SQL Server（包括 SQL Azure）的执行策略。它是感知的可以重试异常类型，并且具有最大重试，重试等之间的延迟合理的默认值。

配置你的上下文的选项时指定的执行策略。这是通常位于 `OnConfiguring` 方法将派生的上下文，或在 `Startup.cs` ASP.NET Core 应用程序。

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

## 自定义执行策略

没有一种机制来注册你自己，如果你想要更改任何默认值的自定义执行策略。

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

## 执行策略和事务

需要能够播放失败重试块中的每个操作失败将自动重试执行策略。启用重试后，通过 EF 核心执行每个操作将成为其自己的可重试操作，即每个查询和每次调用 `SaveChanges()` 将重试作为一个单元如果发生暂时性故障。

但是，如果你的代码启动了事务使用 `BeginTransaction()` 要定义你自己的一组操作需要被视为一个单元，即在事务内的所有内容都将需要播放应发生故障时。如果你尝试执行此操作使用的执行策略时，将收到如下所示的异常。

`InvalidOperationException`：配置的执行策略 `SqlServerRetryingExecutionStrategy` 不支持用户启动事务。使用 `DbContext.Database.CreateExecutionStrategy()` 返回的执行策略在作为可重试单元事务中执行所有操作。

解决方案是使用委托表示的所有内容，需要执行手动调用执行策略。如果发生暂时性故障，则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();
            }
        }
    });
}

```

## 事务提交失败和幂等性问题

一般情况下，连接失败时当前事务将回滚。但是，如果在事务处于断开连接是否将提交所生成的事务的状态为未知。请参阅此[博客文章](#)有关详细信息。

默认情况下，执行策略将重试该操作就像该事务已回滚，但如果它不是这样这将导致异常如果新数据库的状态不兼容，或者可能会导致数据损坏如果操作不依赖于特定的状态，例如，在插入具有自动生成的密钥值的新行时。

有多种与此处理。

### 选项 1-是否执行任何操作（几乎）

事务提交过程中的连接故障的可能性较低，因此它可能是可接受的应用程序以直接失败，如果实际发生这种情况。

但是，你需要避免使用由存储生成的键，以确保添加重复的行而不是引发异常。请考虑使用由客户端生成的 GUID 值或客户端值生成器。

### 选项 2-重新生成应用程序状态

1. 放弃当前 `DbContext`。
2. 创建一个新 `DbContext` 和从数据库中还原你的应用程序的状态。
3. 通知用户，最后一个操作可能不具有已成功完成。

### 选项 3-添加状态验证

对于大多数更改数据库状态的操作则可以添加检查它是否成功的代码。EF 提供一个扩展方法来简化此过程-

`IExecutionStrategy.ExecuteInTransaction`。

此方法开始和提交事务，还接受中的函数 `verifySucceeded` 在事务提交期间发生暂时性错误时调用的参数。

```
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();
}
```

#### 注意

此处 `SaveChanges` 使用调用 `acceptAllChangesOnSuccess` 设置为 `false` 为了避免更改的状态 `Blog` 实体到 `Unchanged` 如果 `SaveChanges` 成功。这样就可以重试相同的操作如果提交失败并回滚事务。

#### 选项 4-手动跟踪的事务

如果你需要使用由存储生成的键或需要不依赖于执行的操作中泛型的方式来处理提交故障的每个事务无法分配提交失败时检查的 ID。

1. 将表添加到用于跟踪事务的状态的数据库。
2. 将行插入表中的每个事务的开头。
3. 如果连接失败的提交，检查数据库中的相应行存在。
4. 如果提交成功，删除相应的行，以避免表的增长。

```
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();
}
```

#### 注意

请确保用于验证的上下文具有定义为连接是可能会在验证期间再次失败，如果在事务提交过程失败的执行策略。

# 测试

2018/1/23 • 1 min to read • [Edit Online](#)

你可能希望使用类似于连接真实数据库的方式来测试组件，但不产生实际数据库 I/O 操作的开销。

可通过以下两种主要方法执行此操作：

- 通过 [SQLite 内存中模式](#)，可针对行为类似于关系数据库的提供程序编写有效测试。
- [InMemory 提供程序](#)是一个具有极少依赖项的轻量提供程序，但其行为不会始终与关系数据库相似。

# 使用 SQLite 测试

2018/1/23 • 3 min to read • [Edit Online](#)

SQLite 具有内存中模式，您可以使用 SQLite 来编写针对关系数据库，而无需实际的数据库操作开销的测试。

## 提示

你可以查看这篇文章[示例](#)GitHub 上

## 示例测试方案

请考虑以下允许应用程序代码执行与博客某些操作的服务。它在内部使用 `DbContext` 连接到 SQL Server 数据库。它会有用要交换此上下文，以便我们可以编写高效测试此服务，而无需修改代码，或执行大量工作来创建测试连接到内存中 SQLite 数据库上下文的双精度。

```
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();
        }
    }
}
```

## 准备你的上下文

### 避免配置两个数据库提供程序

在测试中要从外部配置要使用 InMemory 提供程序的上下文。如果你要配置数据库提供程序通过重写

`OnConfiguring` 在您的上下文，然后你需要添加一些条件的代码，以确保你仅配置数据库提供程序，如果其中一个不已配置。

#### 提示

如果你使用的 ASP.NET 核心, 则应不需要此代码由于数据库提供程序配置 (会在 Startup.cs) 的上下文之外。

```
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

若要启用针对不同的数据库的测试的最简单方法是修改你公开接受一个构造函数的上下文

`DbContextOptions<TContext>`。

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

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

#### 提示

`DbContextOptions<TContext>` 告知所有其设置, 例如要连接到的数据库上下文。这是通过在您的上下文中运行 `OnConfiguring` 方法生成的相同对象。

## 编写测试

与此提供程序测试的关键是无法识别要使用 SQLite, 并控制内存中数据库的作用域的上下文的能力。数据库范围的受打开和关闭连接。数据库的连接已打开的持续时间限定。通常您希望干净的数据库为每个测试方法。

```
using BusinessLogic;
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

namespace TestProject.SQLite
{
    [TestClass]
    public class BlogServiceTests
    {
        [TestMethod]
        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("http://sample.com");
    }

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

[TestMethod]
public void Find_searches_url()
{
    // In-memory database only exists while the connection is open
    var connection = new SqlConnection("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 = "http://sample.com/cats" });
            context.Blogs.Add(new Blog { Url = "http://sample.com/catfish" });
            context.Blogs.Add(new Blog { Url = "http://sample.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.AreEqual(2, result.Count());
        }
    }
    finally
    {
        connection.Close();
    }
}

```





# 测试以及 InMemory

2018/1/23 • 3 min to read • [Edit Online](#)

你想要测试使用的，它模拟连接到真实的数据库，而无需实际的数据库操作开销的组件时，InMemory 提供程序非常有用。

## 提示

可在 [GitHub](#) 上查看此文章的[示例](#)。

## InMemory 不是一个关系数据库

EF 核心数据库提供程序不需要是关系数据库。InMemory 旨在作为一般用途数据库进行测试，而不是要模拟关系数据库。

包括一些示例：

- InMemory 将允许你保存将违反引用完整性约束关系数据库中的数据。
- 如果在模型中的属性使用 `DefaultValueSql(string)`，这是一个关系数据库 API，针对 InMemory 运行时将产生任何影响。

## 提示

对于许多测试目的这些差异将不起作用。但是，如果你想要用作测试依据的表现得更像 true 的关系数据库内容，则可以考虑使用 [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与博客某些操作的服务。它在内部使用 `DbContext` 连接到 SQL Server 数据库。将发挥作用交换此上下文，以便我们可以编写高效测试此服务，而无需修改代码，或执行大量工作来创建测试连接到 InMemory 数据库上下文的双精度。

```

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();
        }
    }
}

```

## 准备你的上下文

### 避免配置两个数据库提供程序

在测试中要从外部配置要使用 InMemory 提供程序的上下文。如果你要配置数据库提供程序通过重写

`OnConfiguring` 在你的上下文，然后你需要添加一些条件的代码，以确保你仅配置数据库提供程序，如果其中一个不已配置。

```

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

```

#### 提示

如果你使用的 ASP.NET 核心，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

若要启用针对不同的数据库的测试的最简单方法是修改你公开接受一个构造函数的上下文

`DbContextOptions<TContext>`。

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

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

#### 提示

`DbContextOptions<TContext>` 告知所有其设置, 例如要连接到的数据库上下文。这是通过在您的上下文中运行 `OnConfiguring` 方法生成的相同对象。

## 编写测试

与此提供程序测试的关键是无法识别要使用的 InMemory 提供程序, 并控制内存中数据库的作用域的上下文的能力。通常您希望干净的数据库为每个测试方法。

下面是使用 InMemory 数据库测试类的一个示例。每个测试方法指定一个唯一的数据库名称, 这意味着每个方法都有自己 InMemory 数据库。

#### 提示

若要使用 `.UseInMemoryDatabase()` 扩展方法, 引用的 NuGet 包 `Microsoft.EntityFrameworkCore.InMemory`。

```

using BusinessLogic;
using Microsoft.EntityFrameworkCore;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Linq;

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

            // Run the test against one instance of the context
            using (var context = new BlogggingContext(options))
            {
                var service = new BlogService(context);
                service.Add("http://sample.com");
            }

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

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

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

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

```

# 配置创建的 DbContext

2018/2/13 • 4 min to read • [Edit Online](#)

这篇文章演示用于配置基本模式 `DbContext` 通过 `DbContextOptions` 连接到使用特定的 EF 核心提供程序和可选行为的数据库。

## 设计时 DbContext 配置

EF 核心设计时工具如 [迁移](#) 需要能够发现和创建的工作实例 `DbContext` 以收集有关应用程序的实体类型以及它们如何映射到数据库架构的详细信息。此过程可能会自动，只要该工具可以轻松地创建 `DbContext`，它将在配置同样到它如何将配置在运行时的方式。

尽管提供必要的配置信息到任何模式 `DbContext` 可在运行时，需要使用的工具 `DbContext` 在设计时只能处理有限数量的模式。这些内容中的更详细地介绍 [设计时上下文创建](#) 部分。

## 配置 DbContextOptions

`DbContext` 必须具有的实例 `DbContextOptions` 才能执行任何工作。`DbContextOptions` 实例传送配置信息，如：

- 数据库提供程序，若要使用，通常选择通过调用方法，如 `UseSqlServer` 或 `UseSqlite`
- 任何必要的连接字符串或数据库实例中，标识符通常作为参数传递给上述提供程序选择方法
- 任何提供程序级别的可选行为的选择器，通常还链接到提供程序选择方法的调用中
- 任何常规 EF 核心行为选择器，通常链接之后或之前提供程序选择器方法

下面的示例将配置 `DbContextOptions` 若要使用 SQL Server 提供程序，在连接包含 `connectionString` 变量、提供程序级别的命令超时，以及可使在中执行的所有查询 EF 核心行为选择器 `DbContext` [否跟踪](#) 默认情况下：

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

### 注意

提供程序选择器方法和上述其他行为选择器方法是扩展方法在 `DbContextOptions` 或提供程序特定的选项类别。才能访问这些扩展方法，你可能需要具有命名空间 (通常 `Microsoft.EntityFrameworkCore`) 中的作用域以及在项目中包含更多文件包依赖关系。

`DbContextOptions` 可以提供给 `DbContext` 通过重写 `OnConfiguring` 方法或外部通过构造函数自变量。

如果将使用它们，`OnConfiguring` 最后应用，并且可以覆盖提供给构造函数自变量的选项。

### 构造函数自变量

使用构造函数的上下文代码：

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

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

#### 提示

`DbContext` 的基构造函数还接受非泛型版本的 `DbContextOptions`，但对于具有多个上下文类型应用程序不建议使用非泛型版本。

应用程序代码以初始化从构造函数自变量：

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

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

## OnConfiguring

使用上下文代码 `OnConfiguring`：

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

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

应用程序代码以初始化 `DbContext` 使用 `OnConfiguring`：

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

#### 提示

此方法不会将自身添加到测试，除非测试以完整的数据库为目标。

## 使用依赖关系注入 DbContext

EF 核心支持使用 `DbContext` 与依赖关系注入容器。`DbContext` 类型可通过使用添加到服务容器 `AddDbContext<TContext>` 方法。

`AddDbContext<TContext>` 将这两种你的 `DbContext` 类型，`TContext`，和相应 `DbContextOptions<TContext>` 可用于从服务容器的注入。

请参阅[详细阅读](#)下面有关依赖关系注入的其他信息。

添加 `DbContext` 依赖关系注入到：

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

这要求添加[构造函数参数](#)为所接受的 `DbContext` 类型 `DbContextOptions<TContext>`。

上下文代码：

```
public class BlogggingContext : DbContext
{
    public BlogggingContext(DbContextOptions<BlogggingContext> 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 BlogggingContext _context;

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

    ...
}
```

(通常不直接, 使用服务提供商处) 的应用程序代码：

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

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

## 详细阅读

- 读取[首先使用 ASP.NET Core](#)有关 ASP.NET 核心中使用 EF 的详细信息。
- 读取[依赖关系注入](#)若要了解有关使用 DI 详细信息。
- 读取[测试](#)有关详细信息。



# 从 EF 核心 1.0 RC1 升级到 1.0 RC2

2018/3/1 • 5 min to read • [Edit Online](#)

本文提供了将用到 RC2 RC1 包生成应用程序迁移指南。

## 包名称和版本

RC1, RC2, 我们从更改"实体框架 7"为"实体框架 Core"。你可以阅读更多有关的更改的原因由 [Scott Hanselman 的此博客文章](#)。由于此更改, 我们包名称更改从 `EntityFramework.*` 到 `Microsoft.EntityFrameworkCore.*` 和我们版本的 `7.0.0-rc1-final` 到 `1.0.0-rc2-final` (或 `1.0.0-preview1-final` 程序工具)。

你将需要完全删除 **RC1** 包, 然后安装 **RC2** 的。下面是一些常见的包的映射。

RC1 包	RC2 等效项
EntityFramework.MicrosoftSqlServer 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.SqlServer 1.0.0-rc2-final
EntityFramework.SQLite 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.Sqlite 1.0.0-rc2-final
EntityFramework7.Npgsql 3.1.0-rc1-3	Npgsql.EntityFrameworkCore.Postgres
EntityFramework.SqlServerCompact35 7.0.0-rc1-final	EntityFrameworkCore.SqlServerCompact35 1.0.0-rc2-final
EntityFramework.SqlServerCompact40 7.0.0-rc1-final	EntityFrameworkCore.SqlServerCompact40 1.0.0-rc2-final
EntityFramework.InMemory 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.InMemory 1.0.0-rc2-final
EntityFramework.IBMDDataServer 7.0.0-beta1	Rc2 尚不可用
EntityFramework.Commands 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.Tools 1.0.0-preview1-final
EntityFramework.MicrosoftSqlServerDesign 7.0.0-rc1-final	Microsoft.EntityFrameworkCore.SqlServer.Design 1.0.0-rc2-final

## 命名空间

命名空间从包名称, 以及更改 `Microsoft.Data.Entity.*` 到 `Microsoft.EntityFrameworkCore.*`。你可以处理此更改与查找/替换的 `using Microsoft.Data.Entity` 与 `using Microsoft.EntityFrameworkCore`。

## 命名约定更改的表

我们在 RC2 中所用的一个重要的功能变化是使用的名称 `DbSet<TEntity>` 属性为给定实体为表名它将映射到, 而不是只是类的名称。你可以阅读更多有关这一更改[相关的公告问题](#)。

对于现有 RC1 应用程序, 我们建议将以下代码添加到开始你 `OnModelCreating` 方法以保持 RC1 命名策略:

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

如果你想要采用新的命名策略，我们建议成功完成的升级步骤然后删除代码并创建要应用的表的迁移的其余部分将重命名。

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

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

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

在 RC2，你可以删除对的调用 `AddEntityFramework()`，`AddSqlServer()`，等等。：

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

你还需要添加一个构造函数，为您派生的上下文，它接受上下文选项并将其传递给基构造函数。这被必须的因为我们删除了一些在后台 snuck 在可怕幻数：

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

## IServiceProvider 传入

如果必须通过的 RC1 代码 `IServiceProvider` 到上下文，这现在已移到 `DbContextOptions`，而不是单独的构造函数参数。使用 `DbContextOptionsBuilder.UseInternalServiceProvider(...)` 设置服务提供商。

### 正在测试

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。请参阅更新[测试文](#)，以执行此操作使用 RC2 的示例。

### 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 应用并且想 EF 若要解决应用程序服务提供商提供的内部服务，则的重载 `AddDbContext`，可以对此进行配置：

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

#### 警告

我们建议允许 EF 内部管理其自身的服务，除非你有其他原因将内部 EF 服务合并到你的应用程序服务提供商。你可能想要执行此操作的主要原因是使用你的应用程序服务提供商来替换 EF 在内部使用的服务

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

如果你以前使用 `dnx ef` 对于 ASP.NET 5 项目的命令，这些现在已移到 `dotnet ef` 命令。相同的命令语法仍然适用。你可以使用 `dotnet ef --help` 语法信息。

注册命令的方式已更改 RC2, 由于.NET CLI 被取代的 DNX 中。在现在注册命令 `tools` 主题中 `project.json` :

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

#### 提示

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 核心项目 (这不支持在 RC1 中) 的 EF 命令。你仍需要注册中的命令 `tools` 部分 `project.json` 要这样做。

## 包管理器命令需要 PowerShell 5

如果在 Visual Studio 中的包管理器控制台中使用实体框架命令, 你将需要确保已安装的 PowerShell 5。这是在下一个版本中将删除的临时要求 (请参阅[发出 #5327](#)有关详细信息)。

## Project.json 中使用"导入"

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

在添加 EF, NuGet 还原将显示此错误消息:

```
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 net451 + win8"的可移植配置文件。此强制将匹配此二进制文件的 NuGet 提供作为.NET standard 兼容框架, 即使它们不是。虽然"可移植 net451 + win8"不是 100%兼容.NET 标准, 它是兼容的 PCL 从转换为.NET 标准。EF 的依赖关系最终将升级到.NET Standard 时, 可能会删除导入。

数组语法中, 可以将多个框架添加到"导入"。如果将其他库添加到你的项目, 则可能需要采用其他导入。

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

请参阅[发出 #5176](#)。

# 从 EF 核心 1.0 RC2 升级到 RTM

2018/3/23 • 3 min to read • [Edit Online](#)

本文提供了将用到 1.0.0 RC2 包生成应用程序迁移指南 RTM。

## 包版本

你通常将安装到应用程序的最上层包的名称未 RC2 和 RTM 之间发生更改。

你需要升级到 **RTM** 版本，安装的程序包：

- 运行时包 (例如 `Microsoft.EntityFrameworkCore.SqlServer`) 更改，不再是 `1.0.0-rc2-final` 到 `1.0.0`。
- `Microsoft.EntityFrameworkCore.Tools` 包更改从 `1.0.0-preview1-final` 到 `1.0.0-preview2-final`。请注意，工具仍预发行版。

## 现有的迁移可能需要添加的 `maxLength`

在 RC2，在迁移中的列定义如下所示 `table.Column<string>(nullable: true)` 和列的长度在我们将存储迁移背后的代码在某些元数据中进行查找。在 RTM，长度现在包含在基架代码

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

已在使用 RTM 之前基架的任何现有迁移不会 `maxLength` 指定参数。这意味着将使用数据库支持的最大长度 (`nvarchar(max)` SQL Server 上)。这可能是相当不错的某些列，但键外，键的一部分的列或索引需要更新，以包含最大长度。按照约定，450 是最大长度用于密钥外，键和索引列。如果模型中，具有显式配置长度，然后应改为使用该长度。

### ASP.NET 标识

此更改会影响使用 ASP.NET 标识，并且已从预先创建的项目的 RTM 项目模板。项目模板包括用于创建数据库的迁移。必须编辑此迁移，以指定最大长度为 `256` 为以下列。

- **AspNetRoles**
  - 名称
  - NormalizedName
- **AspNetUsers**
  - 电子邮件
  - NormalizedEmail
  - NormalizedUserName
  - UserName

若要进行此更改将导致以下异常时初始迁移已应用到数据库。

```
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 核心：project.json 中删除"导入"

如果你已面向.NET 核心与 RC2, 需要添加 `imports` 到临时的解决方法不支持.NET 标准的 EF 核心依赖关系的某些 `project.json`。现在即可移除这些。

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

#### 注意

截至版本 1.0 RTM, .NET 核心 SDK 不再支持 `project.json` 或开发.NET 核心应用程序使用 Visual Studio 2015。我们建议你从 [project.json 迁移到 csproj](#)。如果使用 Visual Studio, 我们建议你升级到 [Visual Studio 2017](#)。

## UWP: 添加绑定重定向

尝试运行 EF 命令在通用 Windows 平台 (UWP) 项目导致以下错误:

```
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.
```

你需要手动将绑定重定向添加到 UWP 项目。创建名为的文件 `App.config` 在项目根文件夹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 核心 2.0

2018/3/1 • 9 min to read • [Edit Online](#)

## 对所有应用程序都通用的过程

更新现有应用程序到 EF 核心 2.0 可能还需要：

1. 升级到另一个支持 .NET 标准 2.0 应用程序的目标 .NET 平台。请参阅[支持的平台](#)有关详细信息。
2. 标识的提供程序与 EF 核心 2.0 兼容的目标数据库。请参阅[EF 核心 2.0 需要 2.0 版数据库提供程序](#)下面。
3. 升级到 2.0 所有 EF 核心包（运行时和工具）。请参阅[安装 EF 核心](#)有关详细信息。
4. 进行任何必要的代码更改，以弥补的重大更改。请参阅[的重大更改](#)下面部分以了解更多详细信息。

## ASP.NET 核心应用程序

1. 请参阅尤其[初始化应用程序的服务提供程序的新模式](#)如下所述。

### 提示

此新模式时更新应用程序迁移到 2.0 强烈建议和 Entity Framework 核心迁移等的产品功能使起作用所需的采用率。其他常见的替代方法是实现 `IDesignTimeDbContextFactory<TContext>`。

2. 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 核心 2.0 升级到 ASP.NET 核心 2.0。有关升级到 2.0 的 ASP.NET Core 应用程序的详细信息，请参阅[有关该主题的 ASP.NET 核心文档](#)。

## 重大更改

我们执行了显著改进我们的现有的 Api 和 2.0 中的行为的机会。有几个可能需要修改现有应用程序代码的改进，虽然我们认为对于大多数应用程序的影响将很低，在大多数情况下要求只需重新编译和最小的引导式的更改，以替换已过时 Api。

### 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 核心的设计时逻辑的方式更新的建议的模式为 ASP.NET 核心 web 应用程序。以前在设计时，EF 核心会尝试调用 `Startup.ConfigureServices` 直接才能访问应用程序的服务提供程序。在 ASP.NET 核心 2.0 中，配置初始化外部 `Startup` 类。通常使用 EF 核心应用程序配置中，从访问其连接字符串因此 `Startup` 本身已不再够用。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 核心工具时。

ApplicationContext 上发现了没有无参数构造函数。将无参数构造函数添加到 ApplicationContext 或添加的实现 `IDesignTimeDbContextFactory<ApplicationContext>` 中 ApplicationContext 相同的程序集中

ASP.NET 核心 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 核心以在设计时访问应用程序的服务提供程序。如果你正在升级 ASP.NET Core 1.x 应用程序，你将需要更新你 `Program` 类如下所示。

```

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();
    }
}

```

## IDbContextFactory renamed

为了支持不同的应用程序模式，并为用户提供更好地控制如何其 `DbContext` 使用在设计时，我们，在过去，提供了 `IDbContextFactory<TContext>` 接口。EF 核心工具将在设计时发现你的项目中的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一般名，它会误导某些用户尝试重新将其用于其他 `DbContext` -创建方案。它们时 EF 工具然后尝试在设计时使用其实现感到措手不及和导致等命令 `Update-Database` 或 `dotnet ef database update` 失败。

以便进行通信的此接口的强设计时语义，我们已重命名到 `IDesignTimeDbContextFactory<TContext>`。

为 2.0 版本 `IDbContextFactory<TContext>` 仍存在，但标记为过时。

## DbContextFactoryOptions removed

由于上面所述的 ASP.NET 核心 2.0 更改，我们发现， `DbContextFactoryOptions` 已不再需要在新 `IDesignTimeDbContextFactory<TContext>` 接口。以下是应改为使用你选择的选项。

DBCONTEXTFACTORYOPTIONS	替代项
ApplicationBasePath	AppContext.BaseDirectory
ContentRootPath	Directory.GetCurrentDirectory()
EnvironmentName	Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT")

## 设计时更改的工作目录

ASP.NET 核心 2.0 更改也需要使用的工作目录 `dotnet ef` 为了符合运行你的应用程序时，由 Visual Studio 使用的工作目录。这样的一个明显的副作用是该 SQLite 像以前一样，文件名现相对于项目目录而不是输出目录。

## EF 核心 2.0 需要 2.0 版数据库提供程序

为使用 EF 核心 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 核心 2.0。

在 SQL Server 和 SQLite 提供程序交付的 EF 团队和 2.0 版本将可作为 2.0 版的一部分发布。开放源代码第三方提供程序 [SQL Compact](#)，[PostgreSQL](#)，和 [MySQL](#) 正在更新为 2.0。对于所有其他提供程序，请联系提供程序编写器。

## 日志记录和诊断事件已发生了更改

注意：这些更改不应影响大多数应用程序代码。

消息发送到事件 Id `ILogger` 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` 事件现在使用相同的事件 ID 名称作为相应 `ILogger` 消息。事件负载是派生自的所有名义类型 `EventData`。

事件 Id、负载类型和类别均记录在 `CoreEventId` 和 `RelationalEventId` 类。

Id 还从移动 `Microsoft.EntityFrameworkCore.Infrastructure` 到新 `Microsoft.EntityFrameworkCore.Diagnostics` 命名空间。

## EF 核心关系元数据 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。这有助于简化较低级别的元数据 API，从而始终通过调用 `.Relational`（而不是 `.SqlServer`、`.Sqlite` 等）来访问常见关系元数据概念。例如，1.1.x 代码如下：

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

现在应编写如下：

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

而不是使用等方法 `ForSqlServerToTable`，扩展方法现在都可用于编写基于当前的提供程序中使用的条件代码。例如：

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

请注意，此更改仅适用于 Api/元数据为定义\_所有\_关系的提供程序。API 和元数据保持不变，当它是特定于单个提供程序。例如，聚集的索引都是特定于 SQL Sever，因此 `ForSqlServerIsClustered` 和 `.SqlServer().IsClustered()` 仍必须使用。

## 不能控制 EF 服务提供程序

EF 核心使用内部 `IServiceProvider`（即依赖关系注入容器）用于内部实现。应用程序应允许 EF 核心以创建和管理在特殊情况下此提供程序除外。强烈建议考虑删除对任何调用 `UseInternalServiceProvider`。如果应用程序需要调用 `UseInternalServiceProvider`，请考虑 [填写问题](#) 以便我们可以调查其他方法来处理你的方案。

调用 `AddEntityFramework`，`AddEntityFrameworkSqlServer`，等不需要的应用程序代码，除非 `UseInternalServiceProvider` 也被称为。删除任何现有调用 `AddEntityFramework` 或 `AddEntityFrameworkSqlServer` 等 `AddDbContext` 仍应使用相同的方式和前面一样。

## 必须命名为内存中数据库

已删除全局未命名的内存中数据库，而是必须命名为内存中的所有数据库。例如：

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

这创建/使用具有名称 "MyDatabase" 的数据库。如果 `UseInMemoryDatabase` 会再次调用具有相同的名称，则将使用相同的内存中数据库，从而使其可以由多个上下文实例共享。

## 只读的 API 更改

`IsReadOnlyBeforeSave`` 和 `IsReadOnlyAfterSave`，和 `IsStoreGeneratedAlways` 已过时，替



换 `BeforeSaveBehavior` 和 `AfterSaveBehavior`。这些行为应用于任何属性（不只由存储生成的属性），并确定将插入到数据库时应如何使用属性的值（`BeforeSaveBehavior`）或更新现有数据库时行（`AfterSaveBehavior`）。

属性标记为 `ValueGenerated.OnAddOrUpdate`（例如，对于计算列）将默认情况下忽略当前的属性上设置任何值。这意味着无论是否已设置或的被跟踪实体上修改任何值将始终获取由存储生成的值。这可以通过设置为其他更改 `Before\AfterSaveBehavior`。

## 新 `ClientSetNull` 删除行为

在以前版本中，`DeleteBehavior.Restrict` 具有的实体的行为由上下文跟踪的详细信息关闭匹配 `SetNull` 语义。在 EF 核心 2.0 中，新 `ClientSetNull` 作为的默认值为可选关系引入了行为。此行为具有 `SetNull` 语义跟踪的实体和 `Restrict` 创建使用 EF 核心的数据库的行为。在我们的经验，它们是跟踪的实体和数据库最预期/有用的行为。`DeleteBehavior.Restrict` 现在会遵循的跟踪时设置的可选关系的实体。

## 删除的提供程序设计时包

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

这将传播到提供程序的设计时包。这些包（`Microsoft.EntityFrameworkCore.Sqlite.Design`，`Microsoft.EntityFrameworkCore.SqlServer.Design` 等）已删除和其内容合并到主提供程序的包。

若要启用 `Scaffold-DbContext` 或 `dotnet ef dbcontext scaffold` 在 EF 核心 2.0 中，你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" />
```

# Entity Framework 6

2018/1/23 • 1 min to read • [Edit Online](#)

Entity Framework 6 (EF6) 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。它作为 .NET Framework 3.5 SP1 和 Visual Studio 2008 SP1 的一部分, 于 2008 年初次发布。从 EF4.1 版本开始, 它已经作为 [EntityFramework NuGet 包](#) 发布 - 目前是 NuGet.org 上最受欢迎的包之一。

目前, [msdn.com/data/ef](https://msdn.com/data/ef) 中提供有 Entity Framework 6 文档。