# Relationships, navigation properties and foreign keys

## Creating and modifying relationships

在一个外键关联中，当你改变关系时，一个依赖对象的状态会从EntityState.Unchanged改变到EntityState.Modified。在一个独立关系中，改变关系不会更新依赖对象的状态。

以下实例演示如何使用外键属性和导航属性来关联相关的对象。在外键关联中，可以使用方法来更新，创建或修改关系。在独立关联中，无法使用外键属性。

* 通过给一个外键属性赋予一个新的值：

course.DepartmentID = newCourse.DepartmentID;

* 下例代码通过设置外键为**null**来移除一个关系。注意：这个外键属性必须是可空的。

course.DepartmentID = null;

### 注意：

如果这个引用是在新增的状态（这个例子中的course对象），引用的导航属性不会被一个新的对象的关键值同步，直到SaveChanges被调用。同步不会发生是因为新增对象在他们被保存之前，他们的对象上下文并不包含永久的键(keys)。如果你新增的对象必须在你设置关系之后保持完全的同步，使用以下方法。

* 通过赋予一个新的对象给导航属性。以下代码在一个course和一个department之间创建了一个关系。如果对象被附加(attached)到context，那么course也同时被添加到department.Courses的集合中，并且course对象上相应的外键属性也被设置为department的关键属性值。

course.Department = department;

* 要删除关系，设置导航属性为null。如果使用基于.NET4.0的Entity Framework，需要在设置之前载入关联端。比如：

context.Entry(course).Reference(c => c.Department).Load();

course.Department = null;

从基于.NET 4.5的Entity Framework 5.0开始，不需要载入关联端。

context.Entry(course).Reference(c => c.Department).CurrentValue = null;

* 通过使用ChangeRelationshipState方法来改变两个实体对象之间的特定关系的状态。这个方法最常被用在N-Tire程序和独立关联（不能被用于外键关联上）。同时，要使用这个方法你必须跳转到ObjectContext，如下例所示。

在以下例子中，Instructors 和Course之间存在多对多的关系。调用ChangeRelationshipState方法并传递EntityState.Added参数，让SchoolContext知道一个关系被新增到两个对象之间。

((IObjectContextAdapter)context).ObjectContext.

ObjectStateManager.

ChangeRelationshipState(course, instructor, c => c.Instructor, EntityState.Added);

注意，如果你正在更新（而不是只新增）一个关系，你必须删除在增加新的关系之后删除原来的关系。

((IObjectContextAdapter)context).ObjectContext.

ObjectStateManager.

ChangeRelationshipState(course, oldInstructor, c => c.Instructor, EntityState.Deleted);

## Synchronizing the changes between the foreign keys and navigation properties

当你使用以上描述的方法来更改附加到context中的对象的关系时，Entity Framework需要保持外键，引用和集合的同步。Entity Framework通过代理来为POCO实例自动管理同步（也称关系修复Relationshi Fix-up）。For more information, see [Working with Proxies](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/proxies).

如果你不通过代理来使用POCO实例，你必须确保**DetectChanges** 方法被调用来同步context中相关的对象。注意，以下APIs自动触发一个**DetectChanges** 调用。

* DbSet.Add
* DbSet.AddRange
* DbSet.Remove
* DbSet.RemoveRange
* DbSet.Find
* DbSet.Local
* DbContext.SaveChanges
* DbSet.Attach
* DbContext.GetValidationErrors
* DbContext.Entry
* DbChangeTracker.Entries
* Executing a LINQ query against a DbSet

## Loading related objects

在Entity Framework中，通常使用定义的关联来使用导航属性来载入与返回实例相关的实例。For more information, see [Loading Related Objects](https://docs.microsoft.com/en-us/ef/ef6/querying/related-data).

### 注意：

### 在外键关联中，当你载入一个依赖对象的相关端，会基于当前内存中依赖的外键的值来载入相关的对象。

// Get the course where currently DepartmentID = 2.

Course course2 = context.Courses.First(c=>c.DepartmentID == 2);

// Use DepartmentID foreign key property to change the association.

course2.DepartmentID = 3;

// Load the related Department where DepartmentID = 3

context.Entry(course).Reference(c => c.Department).Load();

在一个独立关联中，一个依赖对象的相关端是基于当前数据库中的外键值来查询。然而，如果关系被修改，并且依赖对象上的引用属性指向了载入到对象上下文中的一个不同原则的对象，Entity Framework会尝试创建客户端中定义的关系。

## Managing concurrency

外键关联和独立关联的并发检查都是基于实例的键和其他定义在模型中的实例的属性。当使用EF Designer创建一个模型时，设置ConcurrencyMode特性为**fixed**来声明应该为属性做并发检查。当时用Code First来定义模型时，在你希望进行并发检查的属性上使用ConcurrencyCheck注释。当使用Code First时你也可以使用TimeStamp注释来指定属性需要进行并发检查。在一个给定的类中可以有一个（且只有一个）timestamp属性。Code First将这个属性映射到数据库中的一个非空字段上。

建议在处理参与并发检查和解决方案的实例的时候总是使用外键关联。

For more information, see [Handling Concurrency Conflicts](https://docs.microsoft.com/en-us/ef/ef6/saving/concurrency).

## Working with overlapping Keys

重叠键是组合键，即键中的某些属性同时也是实例中另外一个键的一部分。不能在一个独立关联中拥有重叠键。要改变一个包含重叠键的外键关联，建议修改外键的值，而不是使用对象引用。

# Async query and save

### 注意：

**EF6 Onwards Only –** 这章介绍的特征，APIs等，从Entity Framework 6开始引入。如果使用较早的版本，则某些功能并不支持。

EF6引入了异步查询和保存，通过.NET 4.5引入的[async and await keywords](https://msdn.microsoft.com/library/vstudio/hh191443.aspx)。然而并不是所有的程序都可以从异步模式获得好处，它可以被用在处理长时间运行，网络或者I/O-bound任务(Task)来提升客户端的响应速度和服务器的可伸缩性。

## When to really use async

这节的目的是通过一个简单的方式来观察同步和异步程序执行之间的区别，以介绍异步的概念。它并不是用于介绍任何异步编程提供好处的关键情形。

异步编程主要聚焦于释放正在等待一个不需要占用线程计算时间的操作的托管线程（运行.NET代码的线程）来做其他的工作。比如，当数据库正在处理一个查询时，.NET代码不需要做任何事情。

在客户端程序中（Winform，WPF等），当异步操作被执行的时候，当前线程可以用来保持UI的响应。在服务端程序（ASP.NET等），线程可以被用来处理其他请求 – 这可以减少内存使用以及/或者则兼服务器的吞吐量。

在大部分程序中使用异步并不会有可见的好处甚至是有害的。使用测试，图表和常识来衡量特定情况下使用异步带来的影响。

Here are some more resources to learn about async:

* [Brandon Bray’s overview of async/await in .NET 4.5](https://blogs.msdn.com/b/dotnet/archive/2012/04/03/async-in-4-5-worth-the-await.aspx)
* [Asynchronous Programming](https://msdn.microsoft.com/library/hh191443.aspx) pages in the MSDN Library
* [How to Build ASP.NET Web Applications Using Async](http://channel9.msdn.com/events/teched/northamerica/2013/dev-b337) (includes a demo of increased server throughput)

## Create the model

我们将会使用 [Code First workflow](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database) 来创建模型和生成数据库，然而，异步功能在所有EF模型上都能工作，包括使用EF Designer所创建的。

* Create a Console Application and call it **AsyncDemo**
* Add the EntityFramework NuGet package
  + In Solution Explorer, right-click on the **AsyncDemo** project
  + Select **Manage NuGet Packages…**
  + In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package
  + Click **Install**
* Add a **Model.cs** class with the following implementation

using System.Collections.Generic;

using System.Data.Entity;

namespace AsyncDemo

{

public class BloggingContext : DbContext

{

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

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

}

public class Blog

{

public int BlogId { get; set; }

public string Name { get; set; }

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

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public virtual Blog Blog { get; set; }

}

}

## Create a synchronous program

* Replace the contents of **Program.cs** with the following code

class Program

{

static void Main(string[] args)

{

PerformDatabaseOperations();

Console.WriteLine();

Console.WriteLine("Quote of the day");

Console.WriteLine(" Don't worry about the world coming to an end today... ");

Console.WriteLine(" It's already tomorrow in Australia.");

Console.WriteLine();

Console.WriteLine("Press any key to exit...");

Console.ReadKey();

}

public static void PerformDatabaseOperations()

{

using (var db = new BloggingContext())

{

// Create a new blog and save it

db.Blogs.Add(new Blog

{

Name = "Test Blog #" + (db.Blogs.Count() + 1)

});

db.SaveChanges();

// Query for all blogs ordered by name

var blogs = (from b in db.Blogs

orderby b.Name

select b).ToList();

// Write all blogs out to Console

Console.WriteLine();

Console.WriteLine("All blogs:");

foreach (var blog in blogs)

{

Console.WriteLine(" " + blog.Name);

}

}

}

}

|  |
| --- |
|  |

## Making it asynchronous

class Program

{

static void Main(string[] args)

{

var task = PerformDatabaseOperations();

Console.WriteLine("Quote of the day");

Console.WriteLine(" Don't worry about the world coming to an end today... ");

Console.WriteLine(" It's already tomorrow in Australia.");

task.Wait();

Console.WriteLine();

Console.WriteLine("Press any key to exit...");

Console.ReadKey();

}

public static async Task PerformDatabaseOperations()

{

using (var db = new BloggingContext())

{

// Create a new blog and save it

db.Blogs.Add(new Blog

{

Name = "Test Blog #" + (db.Blogs.Count() + 1)

});

Console.WriteLine("Calling SaveChanges.");

await db.SaveChangesAsync();

Console.WriteLine("SaveChanges completed.");

// Query for all blogs ordered by name

Console.WriteLine("Executing query.");

var blogs = await (from b in db.Blogs

orderby b.Name

select b).ToListAsync();

// Write all blogs out to Console

Console.WriteLine("Query completed with following results:");

foreach (var blog in blogs)

{

Console.WriteLine(" - " + blog.Name);

}

}

}

}

|  |
| --- |
|  |

# Code-based configuration

Entity Framework程序的配置可以通过配置文件（app.config/web.config）或者通过代码。后者被称为基于代码的配置。

配置文件配置在Config File章节中描述。配置文件优先权高于基于代码的配置。换句话说，如果一个配置选项同时在代码和配置文件中设置，则会使用配置文件中的配置。

## Using DbConfiguration

EF6及以上版本的基于代码的配置是通过创建一个System.Data.Entity.Config.DbConfiguration类的子类来实现的。当创建子类时需要遵循以下原则：

* 程序中只创建一个DbConfiguration类。这个类声明程序域范围内的设置。
* 将DbConfiguration类与DbContext类放置在同一个程序集中。
* 给DbConfiguration类一个公开的无参构造函数。
* 通过在这个构造函数中调用DbConfiguration的protected的方法来设置配置选项。

遵循这些原则允许EF来自动发现并使用你的配置。

## Example

public class MyConfiguration : DbConfiguration

{

public MyConfiguration()

{

SetExecutionStrategy("System.Data.SqlClient", () => new SqlAzureExecutionStrategy());

SetDefaultConnectionFactory(new LocalDBConnectionFactory("mssqllocaldb"));

}

}

这个类设置EF使用SQL Azure执行战略 – 自动重试失败的数据库操作 – 并且使用Local DB作为数据库。

## Moving DbConfiguration

有些情况下，可能无法将DbConfiguration类放在DbContext的同一个程序集中。比如，你有两个DbContext类分别定义在不同的程序集中。有两种方式来处理这种情况。

第一个选择是使用配置文件来指定将要使用的DbConfiguration实例。通过设置entityFramework 节点的codeConfigurationType 特性来实现。比如：

<entityFramework codeConfigurationType="MyNamespace.MyDbConfiguration, MyAssembly">

...Your EF config...

</entityFramework>

codeConfigurationType的值必须是你的DbConfiguration类的程序集和命名空间限定名。

第二种方式是在你的context类上设置DbConfigurationTypeAttribute。比如：

[DbConfigurationType(typeof(MyDbConfiguration))]

public class MyContextContext : DbContext

{

}

传递给这个特性的值可以是你的DbConfiguration类的类型，如上所示，或者程序集和命名空间限定名字符串。比如：

[DbConfigurationType("MyNamespace.MyDbConfiguration, MyAssembly")]

public class MyContextContext : DbContext

{

}

## Setting DbConfiguration explicitly

在一些情况中，可能在使用任何DbContext 类型之前就需要用到配置。比如：

* 在不使用context的情况下，使用DbModelBuilder来创建一个模型。
* 使用一些其他的利用一个DbContext 的框架/utility代码，这个context会在你的程序的context被使用之前使用。

在这些情况下，EF无法自动发现配置，你必须做以下其中之一：

* 在配置文件中设置DbConfiguration类型，如上Moving DbConfiguration章节所述。
* 在程序启动的时候调用静态的DbConfiguration.SetConfiguration方法。

## Overriding DbConfiguration

某些情况下你需要重载配置在DbConfiguration中的设置。这通常不是程序开发者要做的，而是不能使用一个继承的DbConfiguration类的第三方提供商和插件。

出于这个原因，EntityFramework允许注册一个事件处理器来修改已经存在的配置（在锁定这个配置之前）。它同时也提供了一个sugar方法专门用于替换EF service locator返回的服务。以下是可能使用的方式：

* 在程序启动（使用EF之前），插件或供应商应该为这个事件注册事件处理方法。（注意必须发生在程序使用EF之前）。
* 事件处理句柄为每个需要替换的服务调用ReplaceService。

比如，要替换IDbConnectionFactory 和 DbProviderService，你需要注册一个类似的处理器：

DbConfiguration.Loaded += (\_, a) =>

{

a.ReplaceService<DbProviderServices>((s, k) => new MyProviderServices(s));

a.ReplaceService<IDbConnectionFactory>((s, k) => new MyConnectionFactory(s));

};

上述代码中 MyProviderServices 和MyConnectionFactory代表你自己的服务实现。

你也可以增加额外的依赖性处理器来实现同样的效果。

Note that you could also wrap DbProviderFactory in this way, but doing so will only affect EF and not uses of the DbProviderFactory outside of EF. For this reason you’ll probably want to continue to wrap DbProviderFactory as you have before.

You should also keep in mind the services that you run externally to your application - for example, when running migrations from the Package Manager Console. When you run migrate from the console it will attempt to find your DbConfiguration. However, whether or not it will get the wrapped service depends on where the event handler it registered. If it is registered as part of the construction of your DbConfiguration then the code should execute and the service should get wrapped. Usually this won’t be the case and this means that tooling won’t get the wrapped service.

# Configuration File Settings

Entity Framework允许在配置文件中指定一些设置。通常EF遵循‘convention over configuration’原则：这篇文章中讨论的所有的设置都有一个默认行为，你只需要在默认值不再满足你的需求时才考虑去更改设置。

## A Code-Based Alternative

所有的这些设置都可以通过代码来实现。从EF6开始，我们引入了[code-based configuration](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/code-based)，它提供了一个集中式的方式来在代码中实现配置。EF6之前，也可以通过代码来实现配置但是需要使用各种APIs来配置不同的区域。配置文件选项允许这些设置可以在部署的时候轻易修改而不需要更新源代码。

## The Entity Framework Configuration Section

从EF 4.1开始你可以使用配置文件中的**appSettings** 节点为一个context设置数据库初始化器。在EF 4.3中我们引入了自定义**entityFramework** 节点来处理新的设置。Entity Framework依然可以识别使用旧格式设置的数据库初始化器，但我们建议尽可能使用新的方式。

**entityFramework** 节点会在你安装了EntityFramework NuGet package之后自动被添加到项目的配置文件中。

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

<configuration>

<configSections>

<!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->

<section name="entityFramework"

type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.3.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />

</configSections>

</configuration>

## Connection Strings

[This page](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/connection-strings)提供更详细的信息关于Entity Framework如何决定将要使用的数据库，包括配置文件中的连接字符串。

连接字符串加在标准的**connectionStrings**元素中，并且不需要**entityFramework**节点。

Code First based模型使用普通的ADO.NET连接字符串。比如：

<connectionStrings>

<add name="BlogContext"

providerName="System.Data.SqlClient"

connectionString="Server=.\SQLEXPRESS;Database=Blogging;Integrated Security=True;"/>

</connectionStrings>

EF Designer Based模型使用特定的EF连接字符串。比如：

<connectionStrings>

<add name="BlogContext"

connectionString=

"metadata=

res://\*/BloggingModel.csdl|

res://\*/BloggingModel.ssdl|

res://\*/BloggingModel.msl;

provider=System.Data.SqlClient

provider connection string=

&quot;data source=(localdb)\mssqllocaldb;

initial catalog=Blogging;

integrated security=True;

multipleactiveresultsets=True;&quot;"

providerName="System.Data.EntityClient" />

</connectionStrings>

## Code-Based Configuration Type (EF6 Onwards)

从EF6开始，你可以为EF指定DbConfiguration来在你的程序中使用[code-based configuration](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/code-based)。大部分情况你不需要指定这个设置，因为EF自动发现你的DbConfiguration。For details of when you may need to specify DbConfiguration in your config file see the **Moving DbConfiguration** section of [Code-Based Configuration](https://docs.microsoft.com/en-us/ef/ef6/fundamentals/configuring/code-based).

## EF Database Providers (EF6 Onwards)

EF6之前，一个数据库供应商的Entity Framework特定的部分必须被包含进来作为核心ADO.NET供应商的一部分。从EF6开始，现在EF的特定部分管理和注册是分离的。

通常你不需要自己去注册供应商。通常当你安装它（EF）的时候就已经完成。

通过在**entityFramework** 节点中的**providers** 子节点中包含一个**provider**元素来注册供应商。供应商注册需要两个必须的特性：

* **invariantName**：这个EF供应商所对应的核心ADO.NET供应商。
* **type** ：EF供应商的实现的程序限定类型名

### 注意：

### 一个程序集限定名是命名空间限定名，之后带一个逗号，再接类型所在的程序集名。可选的，也可以指定程序集版本，文化和public key token。

例子：

<providers>

<provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />

</providers>

## Interceptors (EF6.1 Onwards)

从EF 6.1开始，可以在配置文件中注册拦截器。拦截器允许你在EF执行某些的操作的时候，比如执行数据库查询，打开链接等的时候，执行额外的逻辑。

通过在**entityFramework** 节点中的**interceptors** 子节点中包含一个**interceptor** 元素来注册供拦截器。比如，下例中的配置注册了内置的**DatabaseLogger**拦截器，它会记录所有的数据库操作到控制台。

<interceptors>

<interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework"/>

</interceptors>

### Logging Database Operations to a File (EF6.1 Onwards)

在你希望对一个现存的程序增加日志功能来帮助Debug问题的时候，通过配置文件来注册拦截器非常有用。**DatabaseLogger** 支持将日志记录到一个文件，通过在构造函数参数中指定文件名。

<interceptors>

<interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">

<parameters>

<parameter value="C:\Temp\LogOutput.txt"/>

</parameters>

</interceptor>

</interceptors>

默认情况，这会导致每次程序启动的时候日志文件都会被一个新的文件覆盖。要改成当日志文件存在的时候扩展这个日志文件，可以提供另外一个参数：

<interceptors>

<interceptor type="System.Data.Entity.Infrastructure.Interception.DatabaseLogger, EntityFramework">

<parameters>

<parameter value="C:\Temp\LogOutput.txt"/>

<parameter value="true" type="System.Boolean"/>

</parameters>

</interceptor>

</interceptors>

For additional information on **DatabaseLogger** and registering interceptors, see the blog post [EF 6.1: Turning on logging without recompiling](https://blog.oneunicorn.com/2014/02/09/ef-6-1-turning-on-logging-without-recompiling/).

## Code First Default Connection Factory

配置允许指定一个默认的链接工厂，Code First应该用它来定位一个供context使用的数据库。默认的链接工厂只在没有在配置文件中设置链接字符串的时候才会被使用。

当你安装EF NuGet package的时候，一个默认的链接工厂被注册，它指向SQL Express 或者 LocalDB，取决你安装了哪一个。

要设置一个链接工厂，在**defaultConnectionFactory**元素中声明程序集限定类型名。

例如：

<entityFramework>

<defaultConnectionFactory type="MyNamespace.MyCustomFactory, MyAssembly"/>

</entityFramework>

上例要求自定义的工厂必须需要有一个无参构造函数。如果需要，你可以使用**parameters** 元素指定构造函数参数。

比如，SqlCeConnectionFactory，包含在EF中，要求你提供一个供应商的不变名（invariant name）给它的构造函数作为参数。供应商的不变名声明了你希望使用的SQL Compact的版本。以下配置将会导致context默认使用SQL Compact version 4.0。

<entityFramework>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory, EntityFramework">

<parameters>

<parameter value="System.Data.SqlServerCe.4.0" />

</parameters>

</defaultConnectionFactory>

</entityFramework>

如果你不设置一个默认的链接工厂，Code First 会使用SqlConnectionFactory，指向.\SQLEXPRESS。SqlConnectionFactory也有一个构造函数允许你重写连接字符串的一部分。如果你希望使用另一个SQL Server实例而不是.\SQLEXPRESS，可以使用这个构造函数来设置。

以下例子会导致Code First为context使用**MyDatabaseServer** ，如果没有显式设置连接字符串的话。

<entityFramework>

<defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory, EntityFramework">

<parameters>

<parameter value="Data Source=MyDatabaseServer; Integrated Security=True; MultipleActiveResultSets=True" />

</parameters>

</defaultConnectionFactory>

</entityFramework>

默认情况，假设构造函数参数是字符串类型。你可以使用type特性来改变。

<parameter value="2" type="System.Int32" />

## Database Initializers

数据库初始化器是基于每个-context基础上配置的。可以使用**context** 元素来设置。这个元素使用程序集限定名来声明将要被配置的context。

默认情况，Code First contexts被配置为使用CreateDatabaseIfNotExists 初始化器。**context** 元素上有一个**disableDatabaseInitialization** 特性可以被用来禁止数据库初始化。

比如，以下配置为定义在MyAssembly.dll中的Blogging.BlogContext context禁止数据库初始化。

<contexts>

<context type=" Blogging.BlogContext, MyAssembly" disableDatabaseInitialization="true" />

</contexts>

可以使用**databaseInitializer** 元素来设置一个自定义的初始化器。  
<contexts>

<context type=" Blogging.BlogContext, MyAssembly">

<databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly" />

</context>

</contexts>

构造函数参数使用与默认链接工厂相同的语法。

<contexts>

<context type=" Blogging.BlogContext, MyAssembly">

<databaseInitializer type="Blogging.MyCustomBlogInitializer, MyAssembly">

<parameters>

<parameter value="MyConstructorParameter" />

</parameters>

</databaseInitializer>

</context>

</contexts>

你可以配置一个包含在EF中的泛型的数据库初始化器。**type**特性使用.NET Framework格式用于泛型类型。

比如，如果你在使用Code First Migrations，可以使用MigrateDatabaseToLatestVersion<TContext, TMigrationsConfiguration>初始化器来配置，使得数据库自动被迁移/升级（migrated ）。

<contexts>

<context type="Blogging.BlogContext, MyAssembly">

<databaseInitializer type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[[Blogging.BlogContext, MyAssembly], [Blogging.Migrations.Configuration, MyAssembly]], EntityFramework" />

</context>

</contexts>

# Connection strings and models

Entity Framework程序通常使用一个继承自DbContext的类。这个衍生类将会调用基类DbContext类的其中一个构造函数，来控制：

* context将会如何连接到数据库 – 即，一个连接字符串如果被发现/使用。
* context是否会使用Code First计算一个模型或者载入一个EF Designer创建的模型。
* 额外的高级选项。

## Use Code First with connection by convention

如果你在程序中还没有做任何配置，然后调用DbContext的无参构造函数将会使DbContext运行在Code First模式，使用一个由convention创建的数据库链接。比如：

public class BloggingContext : DbContext

{

public BloggingContext()

// C# will call base class parameterless constructor by default

{

}

}

在这个例子中，DbContext使用衍生的context类的命名空间限定名 - Demo.EF.BloggingContext – 作为数据库名并创建一个连接字符串用于这个数据库，可以是SQL Express 或LocalDB。如果都安装了，将会使用SQL Express。

Visual Studio 2010默认包含SQL Express，Visual Studio 2012以及之后的版本默认包含LocalDB。在安装过程中，EntityFramework NuGet package检查可用的数据库服务器。当通过convention创建一个连接的时候，NuGet package 然后会通过设置Code First的默认数据库服务器来更新配置文件。如果SQL Express正在运行，那么它将会被使用。如果SQL Express不可用，那么将会注册LocalDB 为默认。如果配置文件已经包含一个默认链接工厂的设置，那么配置文件不会被改变。

## Use Code First with connection by convention and specified database name

如果你在程序中还没有做任何配置，然后在DbContext上调用字符串参数构造函数并传递你所希望使用的数据库名，将会导致DbContext运行在Code First模式，使用一个这个连接到这个名字的数据库的由公约所创建的数据库链接。比如：

public class BloggingContext : DbContext

{

public BloggingContext()

: base("BloggingDatabase")

{}

}

在这个例子中，DbContext使用“BloggingDatabase”作为数据库名并创建一个用于这个数据库的连接字符串，使用SQL Expres或者LocalDB。如果都安装了，将会使用SQL Express。

## Use Code First with connection string in app.config/web.config file

你可能会选择在你的app.config或者web.config中设置一个连接字符串。比如：

<configuration>

<connectionStrings>

<add name="BloggingCompactDatabase"

providerName="System.Data.SqlServerCe.4.0"

connectionString="Data Source=Blogging.sdf"/>

</connectionStrings>

</configuration>

这是一种简单的方式来告诉DbContext使用一个SQL Express或者LocalDB上的数据库 – 上例中声明了一个SQL Server Compact Edition数据库。

如果连接字符串的名字与context的名字相同（无论有没有命名空间限定声明），那么DbContext会在调用无参构造函数的时候发现它。如果连接字符串的名字与context的名字不同，那么你可以通过将这个连接字符串的名字传递给DbContext的构造函数来告诉DbContext使用这个连接字符串。比如：

public class BloggingContext : DbContext

{

public BloggingContext()

: base("BloggingCompactDatabase")

{

}

}

或者，可以以 “name=<connection string name>”的形式来传递参数。比如：

public class BloggingContext : DbContext

{

public BloggingContext()

: base("name=BloggingCompactDatabase")

{

}

}

这种方式显式的表示你希望从配置文件中发现连接字符串。如果没找到，会抛出异常。

## Database/Model First with connection string in app.config/web.config file

使用EF Designer创建的模型与Code First方式不同，当程序运行的时候你的模型已经存在，而不是从代码中生成。模型通常以EDMX文件的形式存在在项目中。

Designer会添加一个EF连接字符串到app.config或者web.config文件。这个连接字符串很特别，因为它包含了如果在EDMX文件中找到信息的信息。比如：

<configuration>

<connectionStrings>

<add name="Northwind\_Entities"

connectionString="metadata=res://\*/Northwind.csdl|

res://\*/Northwind.ssdl|

res://\*/Northwind.msl;

provider=System.Data.SqlClient;

provider connection string=

&quot;Data Source=.\sqlexpress;

Initial Catalog=Northwind;

Integrated Security=True;

MultipleActiveResultSets=True&quot;"

providerName="System.Data.EntityClient"/>

</connectionStrings>

</configuration>

EF Designer还会生成代码告诉DbContext使用这个连接字符串。比如：

public class NorthwindContext : DbContext

{

public NorthwindContext()

: base("name=Northwind\_Entities")

{

}

}

DbContext知道去载入已经存在的模型（而不是使用Code First从代码中计算得来），因为连接字符串是一个EF连接字符串，它包含了要使用的model的细节。

## Other DbContext constructor options

DbContext还包含其他构造函数和使用模式来用于更多高级的情形。其中包括：

* 你可以使用DbModelBuilder类来创建一个Code First模型而不用初始化一个DbContext实例。结果是一个DbModel对象。然后当你准备好要创建DbContext实例的时候，你可以把这个DbModel对象传递给DbContext的构造函数。
* 你可以传递一个完整的连接字符串给DbContext，而不只是数据库或连接字符串。默认这个连接字符串是与System.Data.SqlClient provider一起使用。可以通过给context.Database.DefaultConnectionFactory设置一个不同的IConnectionFactory的实现来改变。
* 可以将一个已经存在的DbConnection对象传递给一个DbContext的构造函数。如果链接对象是一个EntityConnection的实例，那么这个链接中所指定的model将会被使用，而不是使用Code First来计算一个model。如果对象是是其他类型的实例 – 比如，SqlConnection - 那么context将会在Code First模式中使用它。
* 你可以传递一个已存在的ObjectContext 给DbContext的构造函数来创建一个包裹了这个已存在的context的DbContext。这可以用于一个使用ObjectContext但又希望在程序的某些部分获得DbContext的好处的现有程序。

# Dependency resolution

从EF 6开始，Entity Framework包含了一个通用的机制来获取所需的服务的实现。即，当EF使用一些接口或者基类的实例的时候，它会请求一个所用的接口或者基类的特定的实现。这是通过IDbDependencyResolver接口来实现的。

public interface IDbDependencyResolver

{

object GetService(Type type, object key);

}

GetService方法通常由EF调用，并由EF或者程序所提供的一个IDbDependencyResolver实现来处理。当被调用时，type参数是所请求的服务的接口或者基类，key对象参数要么是null，或者是一个提供关于所请求的服务的上下文信息的对象。

除非特别声明，任何返回的对象都必须是线程安全的，因为它会被作为一个单例来使用。大部分情况下，返回的这个对象是一个工厂，这个工厂本身必须是线程安全的，但是这个工厂返回的对象不必一定是线程安全的，因为每次使用的时候都会从这个工厂请求一个新的实例。

# Connection management

## Passing Connections to the Context

### Behavior for EF5 and earlier versions

DbContext有两个接收链接的构造函数：

public DbContext(DbConnection existingConnection, bool contextOwnsConnection)

public DbContext(DbConnection existingConnection, DbCompiledModel model, bool contextOwnsConnection)

使用它们有一些限制：

1. 如果你传递了一个打开着的链接给这些构造函数，那么framework第一次尝试使用它的时候回抛一个InvalidOperationException 告诉你不能re-open一个已经打开的链接。
2. contextOwnsConnection 标记用于解释是否在context被销毁的时候同时销毁底层数据链接。但是，无论这个值怎么设置，当context销毁的时候数据链接总是会被关闭。所以如果你有多个使用相同链接的DbContext，无论那个context第一次被销毁都将会关闭链接（相似的，如果你在DbContext中混合了一个ADO.NET链接，DbContext在销毁的时候也会关闭这个链接）。

可以通过传递一个关闭的链接并在context创建之后执行打开链接的代码来解决第一个限制：

public static void TwoDbContextsOneConnection()

{

using (var context1 = new BloggingContext())

{

var conn =

((EntityConnection)

((IObjectContextAdapter)context1).ObjectContext.Connection)

.StoreConnection;

using (var context2 = new BloggingContext(conn, contextOwnsConnection: false))

{

context2.Database.ExecuteSqlCommand(

@"UPDATE Blogs SET Rating = 5" +

" WHERE Name LIKE '%Entity Framework%'");

var query = context1.Posts.Where(p => p.Blog.Rating > 5);

foreach (var post in query)

{

post.Title += "[Cool Blog]";

}

context1.SaveChanges();

}

}

}

第二个限制只是意味着你要避免销毁DbContext对象，直到你已经准备好关闭链接。

### Behavior in EF6 and future versions

在EF 6和将来的版本中DbContext有两个相同的构造函数但不再传递给她们的链接是关闭的。所以以下是可能的：

public static void PassingAnOpenConnection()

{

using (var conn = new SqlConnection("{connectionString}"))

{

conn.Open();

var sqlCommand = new SqlCommand();

sqlCommand.Connection = conn;

sqlCommand.CommandText =

@"UPDATE Blogs SET Rating = 5" +

" WHERE Name LIKE '%Entity Framework%'";

sqlCommand.ExecuteNonQuery();

using (var context = new BloggingContext(conn, contextOwnsConnection: false))

{

var query = context.Posts.Where(p => p.Blog.Rating > 5);

foreach (var post in query)

{

post.Title += "[Cool Blog]";

}

context.SaveChanges();

}

var sqlCommand2 = new SqlCommand();

sqlCommand2.Connection = conn;

sqlCommand2.CommandText =

@"UPDATE Blogs SET Rating = 7" +

" WHERE Name LIKE '%Entity Framework Rocks%'";

sqlCommand2.ExecuteNonQuery();

}

}

同时contextOwnsConnection 标记现在可以控制是否在DbContext销毁的时候关闭和销毁链接。所以在上述例子中当context被销毁的时候链接并没有被关闭，而在之前版本的EF中它可能已经被关闭了。

当然依然可以通过设置contextOwnsConnection 为true或者使用其他构造函数来让DbContext控制链接的状态。

### Note

### There are some additional considerations when using transactions with this new model. For details see [Working with Transactions](https://docs.microsoft.com/en-us/ef/ef6/saving/transactions).

## Database.Connection.Open()

### Behavior for EF5 and earlier versions

在EF5和更早版本中有一个Bug，**ObjectContext.Connection.State**没有被更新来反映底层数据链接的真实状态。比如，如果你执行下例代码你可能会返回一个**Closed** 状态，即使底层数据链接依然是**Open**的。

((IObjectContextAdapter)context).ObjectContext.Connection.State

另外，如果你通过调用Database.Connection.Open()来打开数据库链接，它会在下一次你执行一个查询语句或者调用任何依赖数据库链接的语句（比如，SaveChanges()）之前一直保持打开，但在这之后链接将会被关闭。Context会在下一次请求的时候再重新打开和关闭链接：

public static void DatabaseOpenConnectionBehavior()

{

using (var context = new BloggingContext())

{

// At this point the underlying store connection is closed

context.Database.Connection.Open();

// Now the underlying store connection is open

// (though ObjectContext.Connection.State will report closed)

var blog = new Blog { /\* Blog’s properties \*/ };

context.Blogs.Add(blog);

// The underlying store connection is still open

context.SaveChanges();

// After SaveChanges() the underlying store connection is closed

// Each SaveChanges() / query etc now opens and immediately close

// the underlying store connection

blog = new Blog { /\* Blog’s properties \*/ };

context.Blogs.Add(blog);

context.SaveChanges();

}

}

### Behavior in EF6 and future versions

EF6和之后的版本中，如果代码选择通过调用context.Database.Connection.Open()来打开链接，那么framework将会假设代码希望控制链接的打开关闭，并且不会再自动关闭它。

### 注意：

### 这可能会潜在地导致链接长时间打开，所以需要谨慎使用。

并且代码也别更新过，因此现在ObjectContext.Connection.State可以正确反映数据库链接的状态。

public static void DatabaseOpenConnectionBehavior()

{

using (var context = new BloggingContext())

{

// At this point the underlying store connection is closed

context.Database.Connection.Open();

// Now the underlying store connection is open and the

// ObjectContext.Connection.State correctly reports open too

var blog = new Blog { /\* Blog’s properties \*/ };

context.Blogs.Add(blog);

context.SaveChanges();

// The underlying store connection remains open for the next operation

blog = new Blog { /\* Blog’s properties \*/ };

context.Blogs.Add(blog);

context.SaveChanges();

// The underlying store connection is still open

} // The context is disposed – so now the underlying store connection is closed

}

# Connection resiliency and retry logic

程序连接到数据库服务器总是容易脆弱的断开链接，由于后台错误以及网络不稳定。然而，在一个基于LAN的环境上与一个专用的数据库服务器工作，这些错误就会比较罕见，不太需要额外的逻辑来处理这些错误。随着基于云的数据库服务器比如Windows Azure SQL Database和建立在不可靠的网络上的链接增加，链接中断变得更常发生。这可能是因为云数据库所使用的防御技术来确保服务的可靠，比如链接限制，或者网络的不稳定导致超时和其他传输错误。

链接弹性（Connection Resiliency）指的是EF自动重试任何由于链接中断而导致错误的命令的能力。

## Execution Strategies

链接重试是由一个IDbExecutionStrategy 接口的实现来处理的。IDbExecutionStrategy 的实现负责接收一个操作，如果一个异常发生，判断是否适合重试，如果是则重试它。EF有4个执行战略：

1. **DefaultExecutionStrategy**：这个执行战略不重试任何操作，它是除了SQL Server以外的数据库的默认选项。
2. **DefaultSqlExecutionStrategy：**这是一个内部执行战略，默认被使用。这个战略不做任何重试，但是，它会包裹任何异常来让用户知道他们可能希望开启链接弹性。