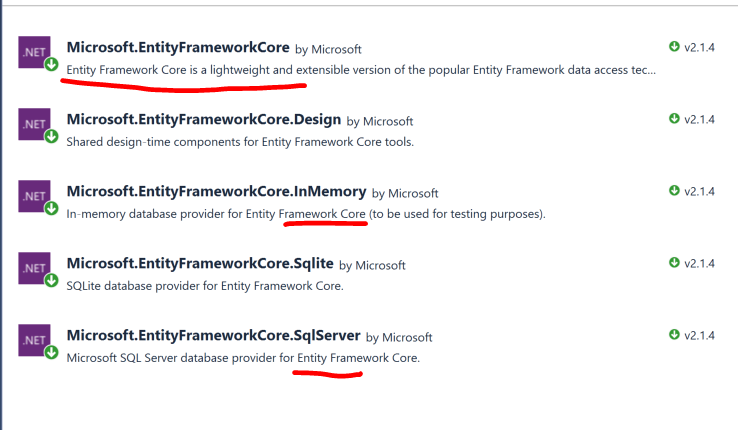
Entity Framework Core

Install Entity Framework Core



**DbContext**

只有两个构造函数， 一个是默认不带参数的， 另一个是 DbContextOptions 或者 DbContextOptions<>

public DbContext([NotNullAttribute] DbContextOptions options);

protected DbContext();

public class ClassRoomDB : DbContext

{

public ClassRoomDB() :base()

{

}

public ClassRoomDB(**DbContextOptions** options) : base(options)

{

}

public ClassRoomDB(**DbContextOptions**<ClassRoomDB> options): base (options)

{

}

protected override void OnConfiguring(**DbContextOptionsBuilder** optionsBuilder)

{

optionsBuilder.UseInMemoryDatabase(Guid.NewGuid().ToString());

//this.Database.EnsureDeleted();

//this.Database.EnsureCreated();

Console.WriteLine("On Configuring: {0}", optionsBuilder.Options.ToString());

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

Console.WriteLine("On ModelCreatiing: {0}", modelBuilder);

}

public DbSet<Student> Students { get; set; }

}

[Table("Students")]

public class Student

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime Birthdate { get; set; }

}

**Main Program:**

DbContextOptionsBuilder optionBuilder = new DbContextOptionsBuilder();

optionBuilder.UseInMemoryDatabase(Guid.NewGuid().ToString());

ClassRoomDB croom = new ClassRoomDB(optionBuilder.Options);

Console.WriteLine("DB Name={0}", croom.Database);

croom.Database.EnsureDeleted();

croom.Database.EnsureCreated();

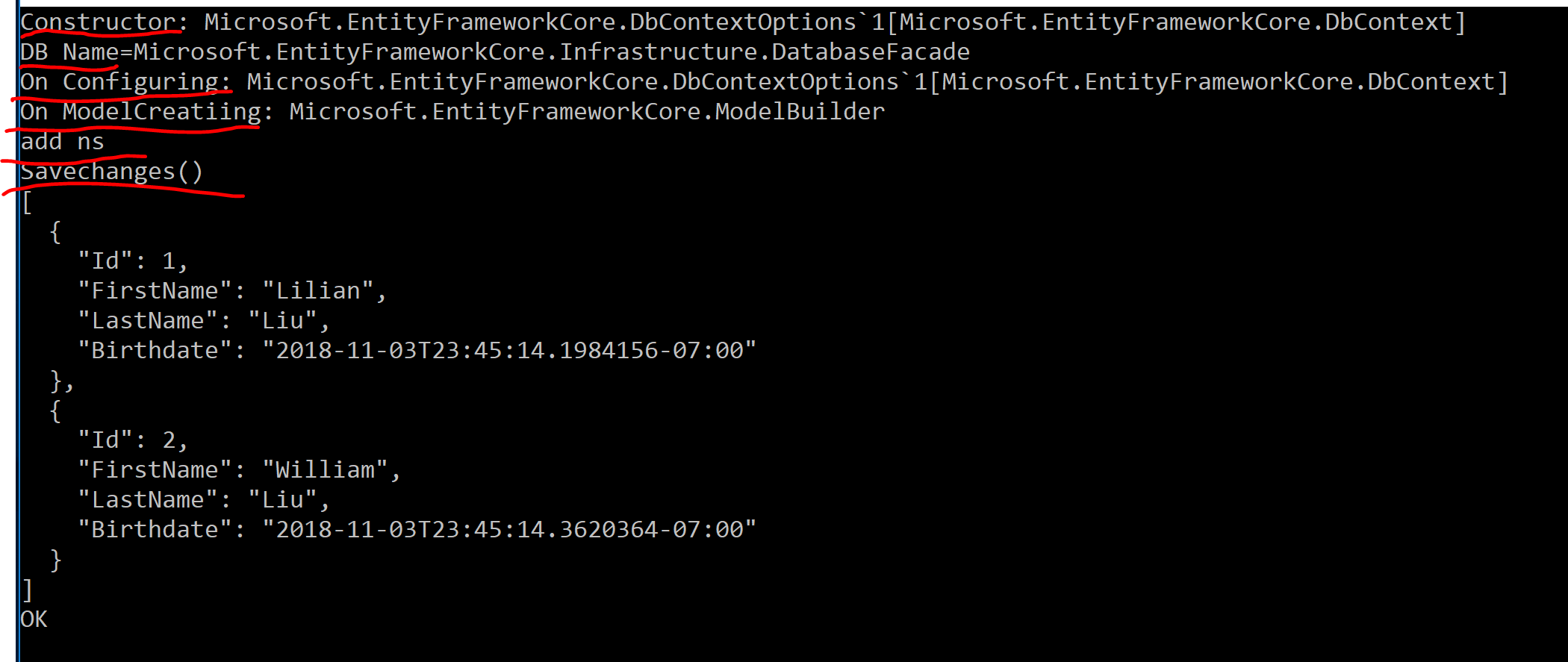
Student ns = new Student { Id = 0, FirstName = "Lilian", LastName = "Liu", Birthdate = DateTime.Now };

croom.Students.Add(ns);

Console.WriteLine("add ns");

croom.SaveChanges();

**执行的顺序：Constructor -> OnConfiguring ->** **OnModelCreating**



Constructor(**DbContextOptions** options)

OnConfiguring(**DbContextOptionsBuilder** optionsBuilder) – 这里在构造函数执行完以后被执行，所以对**DbContextOptions**的设置会覆盖前面构造函数的设置，通常如果是从外部通过构造函数传递 **DbContextOptions ，**那么OnConfiguring() 可以不定义

**UseInMemoryDatabase –** 使用内存数据库，它不用保留到数据库中，而是暂留在内存中

[**https://msdn.microsoft.com/zh-cn/magazine/mt797648**](https://msdn.microsoft.com/zh-cn/magazine/mt797648)

**DbContextOptionsBuilder** optionBuilder = new **DbContextOptionsBuilder**();

optionBuilder.**UseInMemoryDatabase**(**Guid**.NewGuid().ToString());

**Guid**.NewGuid().ToString() – 实际上是随机生成一个数据库名称。

由于内存数据库不需要连接字符串，所以不需要提供其他的参数了。

# 数据点 - 使用 EF Core 及其 InMemory 提供程序生成测试的相关提示

针对触发数据库交互的方法创建自动测试时，有时真的非常希望了解数据库内的运行状况，而有时则会发现数据库交互与测试断言毫不相关。事实证明，在后一种情况中，全新的 EF Core InMemory 提供程序就非常有用。在本文中，我将介绍这一易用工具，并分享一些有关使用 EF Core 创建自动测试的提示和技巧，这些都是我在学习使用此工具时发现的。

如果数据库交互对测试结果无关紧要，那么不必要的数据库调用就可能会造成性能压力，甚至可能会导致测试结果不准确。例如，与数据库通信或删除并重新创建测试数据库所需的时间可能会拖慢测试。另一个令人关切的问题是，如果数据库本身存在问题会怎样。网络延迟或一时的掉线问题可能就会导致测试失败，这只是因为数据库不可用，而不是因为测试断言逻辑出现问题。

我们长期以来都在寻求可以最大限度地降低这些副作用的方法。常见解决方案包括测试虚设和 mock 框架。使用这些模式，可以创建数据存储的内存中表示形式，但设置内存中数据和行为涉及大量工作。另一种方法是，使用比生产目标数据库更为轻型的数据库进行测试。例如，使用 PostgreSQL 或 SQLite 数据库，而不是用于生产数据存储的 SQL Server 数据库。由于有各种可用的提供程序，因此实体框架 (EF) 始终支持通过一个模型使用不同目标数据库。不过，数据库功能的细微差异可能会导致问题发生，所以这种方法并不是百分百有效（但它仍是一种可以一直采用的不错方法）。也可以使用外部工具，如开放源代码 EFFORT 扩展 ([github.com/tamasflamich/effort](http://github.com/tamasflamich/effort))，此扩展巧妙地提供了数据存储的内存中表示形式，省去了测试虚设或 mock 框架所需的设置。EFFORT 适用于从 EF 4.1 到 EF6 的所有版本，但不适用于 EF Core。

EF Core 数据库提供程序有许多。Microsoft 将 SQL Server 和 SQLite 提供程序归到 EntityFrameworkCore API 系列中。还有 SQLCE 和 PostgreSQL 提供程序，分别由 MVP Erik Eilskov Jensen 和 Shay Rojansky 进行维护。此外，市场上还有第三方提供程序。然而，Microsoft 创建了另一种提供程序，它不用保留到数据库中，而是暂留在内存中。这就是 InMemory 提供程序 Microsoft.EntityFrameworkCore.InMemory。在许多测试方案中，可以使用此提供程序快速提供实际数据库的替代数据库。

## 让 DbContext 做好连接 InMemory 提供程序的准备

由于 DbContext 有时用于连接真实的数据存储，有时用于连接 InMemory 提供程序，因此需要将其设置为对提供程序保持灵活性，而不是依赖于任何特定的提供程序。

在 EF Core 中实例化 DbContext 时，必须添加 DbContextOptions 来指定要使用的提供程序和连接字符串（如有需要）。例如，UseSqlServer 和 UseSqlite 要求传入连接字符串，然后各个提供程序会授予对相关扩展方法的访问权限。下面展示了如何直接使用 OnConfiguring 方法在 DbContext 类中这样做，我使用此方法从应用程序配置文件读取连接字符串：

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {

  var settings = ConfigurationManager.ConnectionStrings;

  var connectionString = settings["productionDb"].ConnectionString;

  optionsBuilder.UseSqlServer(connectionString);

 }

不过，更为灵活的模式是将预配置的 DbContextOptions 对象传递给 DbContext 的构造函数：

public SamuraiContext(DbContextOptions<SamuraiContext> options)

    :base(options) { }

EF Core 将把这些预配置的选项传递给基础 DbContext，并为你应用它们。

现在，此构造函数已就位，可以通过使用上下文的逻辑快速指定不同的提供程序（及其他选项，如连接字符串）。

如果要在应用程序中使用某种控制反转 (IoC) 容器（如 StructureMap ([structuremap.github.io](http://structuremap.github.io/)) 或 ASP.NET Core 中内置的服务），可以在配置其他全应用程序 IoC 服务的代码中为上下文配置提供程序。下面的示例在典型 startup.cs 文件中使用 ASP.NET Core 服务：

public void ConfigureServices(IServiceCollection services) {

  services.AddDbContext<SamuraiContext>(

    options => options.UseSqlServer(

      Configuration.GetConnectionString("productionDb")));

  services.AddMvc();

}

在此示例中，SamuraiContext 是继承自 DbContext 的类的名称。我再次使用了 SQL Server，并且我将连接字符串以 productionDb 的名义存储在 ASP.NET Core appsettings.json 文件中。此服务已配置为确定只要类构造函数需要 Samurai­Context 的实例，运行时就不仅应实例化 Samurai­Context，还应传入选项，同时在此方法中声明提供程序和连接字符串。

如果此 ASP.NET Core 应用程序使用我的 SamuraiContext，默认情况下，它现在将会通过 SQL Server 和我的连接字符串完成此操作。不过，由于我在 SamuraiContext 类中内置了灵活性，我还可以创建使用同一 SamuraiContext 的测试，但传入的 DbContextOptions 对象指定改用 InMemory 提供程序，或指定使用与特定测试有关的其他任何选项。

在下一部分中，我将介绍两种涉及 EF Core 的不同测试。第一种（如**图 1** 所示）旨在测试是否发生了正确的数据库交互。也就是说，我真的非常想要此测试与数据库进行交互，因此我将构造 DbContextOptions 来使用 SQL Server 提供程序，但有了以数据库测试版本为目标的连接字符串，我就可以快速创建和删除数据库。

图 1：测试数据库能否按预期插入值

[TestMethod]

  public void CanInsertSamuraiIntoDatabase() {

    var optionsBuilder = new DbContextOptionsBuilder();

    optionsBuilder.UseSqlServer

      ("Server = (localdb)\\mssqllocaldb; Database =

        TestDb; Trusted\_Connection = True; ");

    using (var context = new SamuraiContext(optionsBuilder.Options)) {

      context.Database.EnsureDeleted();

      context.Database.EnsureCreated();

      var samurai = new Samurai();

      context.Samurais.Add(samurai);

      var efDefaultId = samurai.Id;

      context.SaveChanges();

      Assert.AreNotEqual(efDefaultId, samurai.Id);

    }

  }

我使用 EnsureDeleted 和 EnsureCreated 方法来为自己提供全新版本的数据库以供测试，即使没有迁移文件，也可以使用这两种方法。如果有迁移文件，也可以使用 EnsureDeleted 和 Migrate 重新创建数据库。

接下来，我新建一个实体 (samurai)，指示 EF 开始跟踪它，然后记录 SQL Server 提供程序提供的临时键值。调用 SaveChanges 后，我验证 SQL Server 是否已为键应用它自己的数据库生成值，确保此对象确实已正确插入数据库中。

删除并重新创建 SQL Server 数据库可能会影响测试的运行时间。可以在此示例中使用 SQLite，既能更快速地获取相同结果，又能确保测试仍与实际数据库进行交互。此外，还请注意，与 SQL Server 提供程序一样，在你向上下文添加实体时，SQLite 也会设置临时键值。

如果方法恰好使用 EF Core，但你希望在不与数据库进行交互的情况下测试方法，这时 InMemory 提供程序就派上用场了。不过，请注意，由于 InMemory 不是数据库，因此无法仿真关系数据库行为的所有特征（例如，引用完整性）。如果这一点对测试非常重要，可能会首选 SQLite 选项或（像 EF Core 文档建议的一样）SQLite 内存中模式，如[bit.ly/2l7M71p](http://bit.ly/2l7M71p) 所述。

下面是我在应用程序中编写过的一种方法，它使用 EF Core 执行查询并返回 KeyValuePair 对象列表：

public List<KeyValuePair<int, string>> GetSamuraiReferenceList() {

  var samurais = \_context.Samurais.OrderBy(s => s.Name)

    .Select(s => new {s.Id, s.Name})

    .ToDictionary(t => t.Id, t => t.Name).ToList();

  return samurais;

}

我想要测试此方法是否真的会返回 KeyValuePair 列表。我不需要查询数据库即可证明这一点。

下面展示了使用 InMemory 提供程序证明这一点的测试（我已将其安装到测试项目中）：

[TestMethod]

  public void CanRetrieveListOfSamuraiValues() {

    \_options = new DbContextOptionsBuilder<SamuraiContext>()

               .UseInMemoryDatabase().Options;

    var context = new SamuraiContext(\_options);

    var repo = new DisconnectedData(context);

    Assert.IsInstanceOfType(repo.GetSamuraiReferenceList(),

                            typeof(List<KeyValuePair<int, string>>));

  }

此测试甚至不需要向数据库的内存中表示形式提供任何示例数据，因为它足以返回 KeyValuePairs 空列表。运行此测试时，EF Core 会确保当 GetSamuraiReferenceList 执行其查询时，提供程序会在内存中分配资源以供 EF 用作执行依据。如果查询成功，测试也会成功。

如果我想测试返回的结果数量是否正确，该怎么办？ 也就是说，我将需要提供数据，从而向 InMemory 提供程序提供源。与测试虚设或 mock 非常相似，这需要创建数据，并将其加载到提供程序的数据存储中。使用测试虚设或 mock 时，可以创建并填充 List 对象，然后对列表执行查询。InMemory 提供程序负责容器。使用 EF 命令进行预填充。InMemory 提供程序还负责使用测试虚设或 mock 时所需的大部分开销和额外编程工作。

例如，**图 2** 展示了在测试与 InMemory 提供程序进行交互前，我用于为其提供源的方法：

图 2：为 EF Core InMemory 提供程序提供源

private void SeedInMemoryStore() {

    using (var context = new SamuraiContext(\_options)) {

      if (!context.Samurais.Any()) {

        context.Samurais.AddRange(

          new Samurai {

            Id = 1,

            Name = "Julie",

          },

          new Samurai {

            Id = 2,

            Name = "Giantpuppy",

        );

        context.SaveChanges();

      }

    }

  }

如果我的内存中数据为空，此方法会添加两个新的 samurai，然后调用 SaveChanges。现在，提供程序可供测试使用。

但如果我的 InMemory 数据存储中有数据（我刚刚实例化了上下文），会怎样呢？ 上下文并不是 InMemory 数据存储。将数据存储看作是 List 对象，也就是说，上下文会根据需要快速创建它。不过，一旦创建，它就会在应用程序的生存期内一直处于内存中。如果运行的是一个测试方法，则不会出现意外情况。但如果运行的是大量测试方法，那么每个测试方法都使用相同的数据集，你可能就不会希望再次填充它。关于这一点还有更多需要了解的，在你了解更多一些代码后，我就可以解释清楚了。

虽然下一个测试有点人为因素在里面，但旨在展示如何使用填充的 InMemory 存储。既然我刚刚已为内存提供两个 samurai，测试会调用同一 GetSamuraiReferenceList 方法，并断言生成的列表中包含两项：

[TestMethod]

  public void CanRetrieveAllSamuraiValuePairs() {

    var context = new SamuraiContext(\_options);

    var repo = new DisconnectedData(context);

    Assert.AreEqual(2, repo.GetSamuraiReferenceList().Count);

  }

你可能已发现，我并没有调用 Seed 方法，也没有创建选项。我已将此逻辑移到测试类构造函数中，所以无需在测试中重复操作。\_options 变量声明为整个类范围的变量：

private DbContextOptions<SamuraiContext> \_options;

  public TestDisconnectedData() {

    \_options =

      new DbContextOptionsBuilder<SamuraiContext>().UseInMemoryDatabase().Options;

    SeedInMemoryStore();

  }

至此，我已将 Seed 方法移入构造函数，你可能会认为（就像我一样）它只会调用一次。但事实并非如此。你知道每个运行的测试方法都会调用测试类构造函数吗？ 老实说，我已忘记这一点，直到我发现测试独自运行就会通过，而一起运行则无法通过。在我开始检查内存中是否有 samurai 数据前，情况确实如此。触发了要调用的 Seed 方法的所有方法都会为同一集合提供源。不管我是在所有测试方法中都调用 Seed 方法，还是只在构造函数中调用一次，都会发生这种情况。无论如何，检查是否已有数据可为我保驾护航。

还有更好的方法来避免发生内存中数据存储冲突问题。使用 InMemory，可以命名数据存储。

如果要让测试方法恢复 DbContextOptions 创建功能，并让每个方法都这样，将唯一名称指定为 UseInMemory 的参数可以确保所有方法使用的是它自己的数据存储。

我删除了整个类范围的 \_options 变量和类构造函数，从而重构了测试类。我改用一种方法，为已命名数据存储创建选项，并为需要将相应名称用作参数的特定数据存储提供源：

private DbContextOptions<SamuraiContext> SetUpInMemory(string uniqueName) {

  var options = new DbContextOptionsBuilder<SamuraiContext>()

                    .UseInMemoryDatabase(uniqueName).Options;

  SeedInMemoryStore(options);

  return options;

}

我将 SeedInMemoryStore 的签名和第一行修改为，对唯一数据存储使用已配置的选项：

private void SeedInMemoryStore(DbContextOptions<SamuraiContext> options) {

  using (var context = new SamuraiContext(options)) {

现在，每个测试方法均使用此方法和唯一名称来实例化 DbContext。下面展示了修订后的 CanRetrieveAllSamuraiValuePairs。唯一的变化是，我现在传入的是新 SetUpInMemory 方法以及唯一数据存储名称。EF 团队建议的便捷模式是使用测试名称作为 InMemory 资源的名称：

[TestMethod]

  public void CanRetrieveListOfSamuraiValues() {

  using (var context =

new SamuraiContext(SetUpInMemory("CanRetrieveListOfSamuraiValues"))) {

    var repo = new DisconnectedData(context);

    Assert.IsInstanceOfType(repo.GetSamuraiReferenceList(),

                            typeof(List<KeyValuePair<int, string>>));

   }

 }

我的测试类中的其他测试方法有自己的唯一数据存储名称。现在你会发现，可借助模式使用唯一一组数据，或跨测试方法共用一组通用数据。当测试向内存中数据存储写入数据时，使用唯一名称可以避免对其他测试产生副作用。请注意，EF Core 2.0 始终要求提供名称，而 EF Core 1.1 则不是，其中参数是可选的。

我还想要分享关于 InMemory 数据存储的最后一项建议。在编写第一个测试时，我指出，向上下文添加对象时，SQL Server 和 SQLite 提供程序均会将临时值插入 Samurai 的键属性。我没有提及的是，提供程序不会重写你自己指定的值。但无论属于上述哪种情况，由于我使用的是默认数据库行为，因此数据库都会用它自己生成的主键值重写此值。不过，使用 InMemory 提供程序，如果你提供键属性值，那么此值会成为数据存储使用的值。如果未提供值，InMemory 提供程序会使用客户端键生成器，其值可充当数据存储分配的值。

我使用的示例来自于我在 Pluralsight 上的“EF Core： 入门”课程 ([bit.ly/PS\_EFCoreStart](http://bit.ly/PS_EFCoreStart))。学习此课程可以详细了解 EF Core 以及如何使用 EF Core 进行测试。本文还随附可供下载的示例代码。

**Julie Lerman** 住在佛蒙特州的丘陵地区，担任 Microsoft 区域主管、Microsoft MVP、软件团队导师和顾问。可以在全球的用户组和会议中看到她对数据访问和其他主题的介绍。她的博客地址是 [*thedatafarm.com/blog*](http://thedatafarm.com/blog)。她是“Entity Framework 编程”及其 Code First 和 DbContext 版本（全都出版自 O’Reilly Media）的作者。通过 Twitter 关注她：[*@julielerman*](https://twitter.com/@julielerman) 并在 [*juliel.me/PS-Videos*](http://juliel.me/PS-Videos) 上观看其 Pluralsight 课程。