重构 ASP.NET 5/EF6 项目和依赖关系注入 - 文章

依赖关系注入（DI）都是关于松耦合的（bit.ly/1TZWVtW）。您从其他位置（理想情况下是类构造函数）请求获取您依赖的类，而不是将这些类硬编码为其他类。这遵循的是显式依赖关系原则，可以更明确地告知类用户此类所需的协作者。这样一来，您还可以在类对象实例有备选配置的情况下构建更灵活的软件，同时这也对编写这种类的自动测试真正有益。我的工作领域就是和 Entity Framework 代码不停地打交道。典型的例子是：在不使用松耦合的情况下编码就是在创建可直接实例化 DbContext 的存储库或控制器。我已经上千次这样做过。实际上，我撰写这篇文章的目标是为了向我在专栏“EF6、EF7 与 ASP.NET 5 组合”（msdn.com/magazine/dn973011）中编写的代码应用我所学到的 DI 知识。例如，在下面的方法中，我就直接实例化了 DbContext：

```
1
2
3
4
5
6
public List<Ninja> GetAllNinjas() {
    using (var context=new NinjaContext())
    {
        return context.Ninjas.ToList();
    }
}
```

由于我是在 ASP.NET 5 解决方案中使用了这种方法，而且 ASP.NET 5 也内置了如此多的 DI 支持，因此 EF 团队的 Rowan Miller 就建议我利用 DI 支持来改进此示例。我一直以来都十分关注这个问题的其他方面，甚至都没有考虑过这一点。所以，我就开始了一点一点地重构此示例，直到我能让流按规定运行。实际上，Miller 曾指导我参考 Paweł Grudzień 在其博文“结合使用 Entity Framework 6 和 ASP.NET 5”（bit.ly/1k4Tt4Y）中编写的完美示例，但我明确表示会转移我的视线，不会简单地复制粘贴这篇博文中的代码。相反，我是独立重构此示例，这样我就能更好地理解流了。最后，我很高兴地发现，我的解决方案与那篇博文中的非常一致。

一直以来，我似乎都觉得控制反转（IoC）和 IoC 容器是有点艰巨的模式。请注意，自我编码以来已有近 30 年，所以我想我并不是唯一一个在心理上从未准备向此模式过渡的有经验开发者。Martin Fowler 在此领域是非常著名的专家。他指出 IoC 具有多重意义，而与 DI（他为了阐述 IoC 这一优点而创造了这个术语）一致的意义则在于明确应用程序的哪一部分可控制创建特定对象。如果没有 IoC，这一直以来都是个难题。

在与 Steve Smith（devig.com）合著 Pluralsight 课程“域驱动设计基础知识”（bit.ly/PS-DDD）

时，我最终接受了使用 StructureMap 库。此库自 2005 年创建以来就成为 .NET 开发者最常用的 IoC 容器之一。说到底，我参与这场游戏有点晚了。在 Smith 的指导下，我能够了解此库的工作原理及其优势，但仍觉得尚未相当熟练地掌握它。所以，在 Miller 提示我后，我就决定要重构我之前的示例，以便利用容器，这样可以更加容易地将对象实例注入需要使用这些实例的逻辑中。

但首先，让我们来谈谈“不要自我重复”

我的类（囊括了之前所示 GetAllNinjas 类）中最初出现的问题是，我在此类的其他方法中重复使用以下 using 代码：

```
1
using(var context=new NinjaContext)
```

如下所示：

```
public Ninja GetOneNinja(int id) {
   using (var context=new NinjaContext())
   {
       return context.Ninjas.Find(id);
   }
}
```

不要自我重复 (DRY) 原则帮助我发现了这一潜在问题。我将把 NinjaContext 实例的创建代码移入构造函数中，并与各种方法共享诸如 _context 之类的变量：

```
1
2
3
4
NinjaContext _context;
public NinjaRepository() {
   _context = new NinjaContext();
}
```

不过，此类应仅以检索数据为重点，却仍在负责确定如何以及何时创建上下文。我想在流中将确定如何以及何时创建上下文的任务上移，只让我的存储库使用注入的上下文。因此，我将再次重构，以便传递其他位置创建的上下文：

```
1
2
3
4
NinjaContext _context;
public NinjaRepository(NinjaContext context) {
   _context = context;
}
```

现在，存储库就可以独立运行了。我无需通过不断清理它来创建上下文。存储库并不关注上下文的配置方式、创建时间或处置时间。这还有助于此类遵循另一项面向对象的原则（即单一责任原则），因为它除了负责提出数据库请求之外，不再负责管理 EF 上下文。在处理存储库类时，我可以查询为重点。我还可以更轻松地进行测试，因为我的测试可以引导这些决策，不会受到以下存储库的掣肘：设计使用方式与我希望在自动测试使用它的方式不一致。

我原来的示例还存在一个问题，就是我将连接字符串硬编码为 DbContext。我刚才也说明了理由，就是因为这“只是个演示”，而且将连接字符串从正在执行的应用（ASP.NET 5 应用程序）移入 EF6 项目过程十分复杂，而我则关注的是其他方面的问题。不过，在我重构此项目时，我将能够利用 IoC 从正在执行的应用程序传递连接字符串。请继续阅读本文，注意介绍此问题的地方。

让 ASP.NET 5 注入 NinjaContext

不过，我该将 NinjaContext 创建代码移入哪里呢？ 答案就是使用存储库的控制器。我当然不想在控制器中引入 EF，以此来将它传递到存储库的新实例中。这会导致混乱局面的产生（如下所示）：

```
1
2
3
4
5
6
7
8
9
10
public class NinjaController : Controller {
   NinjaRepository _repo;
   public NinjaController() {
       var context = new NinjaContext();
       _repo = new NinjaRepository(context);
   }
   public IActionResult Index() {
       return View(_repo.GetAllNinjas());
   }
}
```

同时，我也在强制控制器注意 EF，此代码不存在我刚才在存储库中解决的实例化依赖对象的问题。控制器可直接实例化存储库类。我只想让它使用存储库，而不用担心具体的创建方式和时间或处置时间。就像我在存储库中注入 NinjaContext 实例一样，我想在控制器中注入随时可用的存储库实例。

控制器类中更简洁的代码版本更像是下面这样：

```
1
```

使用 IoC 容器安排创建对象

由于我要处理的是 ASP.NET 5，而不是 StructureMap 中的请求，因此我将利用 ASP.NET 5 内置的 DI 支持。ASP.NET 5 不仅可以注入许多旨在接受对象的新 ASP.NET 类， 还提供可以协调对象去向的服务基础结构（即 IoC 容器）。它还允许您指定将创建和注入的对象的范围（应何时创建和处置对象）。借助内置支持是更简单的入门方法。

在借助 ASP.NET 5 DI 支持以根据需要注入 NinjaContext 和 NinjaRepository 之前，让我们来看看注入 EF7 类是什么样的情况，因为 EF7 内置可关联 ASP.NET 5 DI 支持的方法。属于标准 ASP.NET 5 项目的 startup.cs 类具有称为“ConfigureServices”的方法。您就是在这其中告知应用程序您想如何关联依赖关系的，以便创建适当的对象并将其注入需要使用这些对象的对象中。在下面的方法中，除 EF7 配置以外的其他所有内容均已被排除：

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddEntityFramework()
                   .AddSqlServer()
                   .AddDbContext(options =>
                       options.UseSqlServer(
                       Configuration["Data:DefaultConnection:ConnectionString"]));
}
```

我的项目使用的是基于 EF6 的模型。与我的项目不同，正在执行此配置的项目依赖 EF7。接下来的几个段落介绍了这种代码出现的具体情况。

由于 EntityFramework .MicrosoftSqlServer 已在其 project.json 文件中指定，因此项目引用所有相关的 EF7 程序集。其中的一个程序集 EntityFramework.Core 向 IServiceCollection 提供了

AddEntityFramework 扩展方法，这样我就能添加 Entity Framework 服务了。EntityFramework .MicrosoftSqlServer dll 提供了 AddSqlServer 扩展方法，此方法追加到了 AddEntityFramework 中。这就将 SqlServer 服务打包入 IoC 容器，以便 EF 知道在查找数据库提供程序时使用它。

AddDbContext 是 EF 的核心方法。这种核心方法可向 ASP.NET 5 内置容器添加指定的 DbContext 实例（包含指定的选项）。在其构造函数中发出 DbContext 请求（以及 ASP.NET 5 正在构造的）所有类将会在创建后获得已配置的 DbContext。所以，这种代码将 NinjaContext 添加为一种已知类型，服务将会根据需要进行实例化。此外，代码还指定在构造 NinjaContext 时，应使用配置代码中的字符串（在此示例中，来自 ASP.NET 5 appsettings.json 文件，由项目模板创建）作为 SqlServer 配置选项。由于 ConfigureService 是在启动代码中运行，因此当应用程序中的所有代码都需要 NinjaContext，但又无任何特定实例提供时，ASP.NET 5 会使用指定的连接字符串实例化并传递新的 NinjaContext 对象。

这就是 EF7 内置的实用功能。遗憾的是，EF6 不具有其中任何一项功能。但既然您知道服务的工作原理，向应用程序服务添加 EF6 NinjaContext 的模式应该就是有意义的。

添加不是专为 ASP.NET 5 构建的服务

除了可以添加与 ASP.NET 5 兼容的服务（包含实用的扩展方法，如 AddEntityFramework 和 AddMvc）之外，还可以添加其他依赖关系。IServicesCollection 接口提供了普通的 Add 方法，以及一系列用于指定所添加服务的生存期的方法： AddScoped、AddSingleton 和 AddTransient。我将针对我的解决方案主要介绍 AddScoped，因为它将请求的实例的生存期限定为 MVC 应用程序（我想在其中使用我的 EF6Model 项目）中每个 HTTP 请求的生存期。此应用程序不会尝试跨请求共享实例。这将会在每个控制器操作中创建和处置我的 NinjaContext，从而模拟我原来要实现的目标，因为每个控制器操作均响应一个请求。

请注意，我有两个需要注入对象的类。NinjaRepository 类需要注入 NinjaContext 对象，而 NinjaController 类则需要注入 NinjaRepository 对象。

在 startup.cs ConfigureServices 方法中，我从添加以下代码入手：

```
1
2
services.AddScoped<NinjaRepository>();
services.AddScoped<NinjaContext>();
```

现在，我的应用程序已注意到这些类型，将在另一个类型的构造函数提出请求时，实例化这些类型。

当控制器构造函数在寻找要作为参数传递的 NinjaRepository 时：

```
1
2
3
public NinjaController(NinjaRepository repo) {
        _repo = repo;
    }
```

但什么内容都没有传递，服务将快速创建 NinjaRepository。这就被称为“构造函数注入”。如果 NinjaRepository 需要 NinjaContext 实例，但什么内容都没有传递，那么服务也会知道进行实例化。

还记得我之前指出的 DbContext 中获取的连接字符串吗？ 现在，我可以告知构造 NinjaContext 的 AddScoped 方法这个连接字符串。我将再次在 appsetting.json 文件中添加此字符串。下面就是此文件中的相应部分：

```
1
2
3
4
5
6
7
"Data": {
       "DefaultConnection": {
           "NinjaConnectionString":
           "Server=(localdb)\\mssqllocaldb;Database=NinjaContext;
           Trusted_Connection=True;MultipleActiveResultSets=true"
       }
   }
```

请注意，JSON 不支持自动换行，因此以 Server= 开头的字符串无法在 JSON 文件中自动换行。此处进行自动换行只是为了方便您阅读。

我已将 NinjaContext 构造函数修改为接收连接字符串，并在 DbContext 重载中使用此字符串，而这也会接收连接字符串：

```
1
2
public NinjaContext(string connectionString):
       base(connectionString) { }
```

现在，我可以告知 AddScoped，在发现 NinjaContext 后，应使用相应的重载构造它，同时传递 appsettings.json 中的 NinjaConnectionString：

```
1
2
3
services.AddScoped
(serviceProvider=>new NinjaContext
   (Configuration["Data:DefaultConnection:NinjaConnectionString"]));
```

在进行这最后一项更改之后，我重构的解决方案现在就能从头到尾正常运行了。启动逻辑将应用设置为注入存储库和上下文。在应用路由到默认控制器（所使用的存储库使用上下文）后，不仅会快速创建所需的对象，还会检索数据库中的数据。我的 ASP.NET 5 应用程序利用其内置 DI 与我在其中使用 EF6 构建模型的旧程序集进行交互。

旨在提高灵活性的接口

还可以进行最后一项改进，就是利用接口。如果可以使用不同版本的 NinjaRepository 或 NinjaContext 类，我可能会从头到尾实现接口。由于我无法预见是否需要在 NinjaContext 上使用变体，因此我将只为存储库类创建接口。

如图 1 所示，NinjaRepository 现在实现 INinjaRepository 协定。

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
public interface INinjaRepository
{
    List<Ninja> GetAllNinjas();
}
public class NinjaRepository : INinjaRepository
{
    NinjaContext _context;
    public NinjaRepository(NinjaContext context) {
        _context = context;
    }
    public List<Ninja> GetAllNinjas() {
        return _context.Ninjas.ToList();
    }
}
```

ASP.NET 5 MVC 应用程序中的控制器现在使用 INinjaRepository 接口，而不是 NinjaRepository 的具体实现：

```
1
2
3
4
```

```
5
6
7
8
9
public class NinjaController : Controller {
    INinjaRepository _repo;
    public NinjaController(INinjaRepository repo) {
        _repo = repo;
    }
    public IActionResult Index() {
        return View(_repo.GetAllNinjas());
    }
}
```

我已修改了 NinjaRepository 的 AddScoped 方法，以告知 ASP.NET 5 无论何时需要使用接口，使用相应的实现（当前是 NinjaRepository）：

```
1
services.AddScoped<INinjaRepository, NinjaRepository>();
```

如果需要使用新版本，或者我要在其他应用程序中使用接口的不同实现，那么我可以将 AddScoped 方法修改为使用正确的实现。

在实践中学习，而非仅仅复制粘贴

我非常感激 Miller 有礼貌地为我提出了重构解决方案的挑战。从我所撰写的内容来看，我的重构之路自然不像看起来那么顺利。由于我并不是简单地复制其他人的解决方案，因此我在一开始就做错了几个地方。了解所出现的问题并编写出正确的代码，不仅让我取得了成功，还在很大程度上帮助我了解 DI 和 IoC。我希望我撰写的内容可以为您带来同样的帮助，以免您像我一样在黑暗中摸索。

Julie Lerman 是 Microsoft MVP、.NET 导师和顾问，住在佛蒙特州的山区。您可以在全球的用户组和会议中看到她对数据访问和其他 .NET 主题的演示。她的博客地址是 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 并在 juliel.me/PS-Videos 上观看其 Pluralsight 课程。

衷心感谢以下技术专家对本文的审阅： Steve Smith
Steve Smith (@ardalis) 是一位对构建优质软件热忱执着的企业家和软件开发者。Steve 已发布多门 Pluralsight 课程，涉及 DDD、SOLID、设计模式和软件体系结构等方面。他身兼数职（Microsoft MVP、作者、导师和培训师），经常在开发者大会上发言。请访问 ardalis.com，看看您的团队或项目可以从 Steve 那儿获得哪些帮助。

合作联系

Email：bd@Jobbole.com

QQ： 2302462408 （加好友请注明来意）

更多频道

小组 - 好的话题、有启发的回复、值得信赖的圈子
头条 - 分享和发现有价值的内容与观点
相亲 - 为IT单身男女服务的征婚传播平台
资源 - 优秀的工具资源导航
翻译 - 翻译传播优秀的外文文章
文章 - 国内外的精选文章
设计 - UI,网页，交互和用户体验
iOS - 专注iOS技术分享
安卓 - 专注Android技术分享
前端 - JavaScript, HTML5, CSS
Java - 专注Java技术分享
Python - 专注Python技术分享