**关于DbContext**

这不是第一篇写关于如何管理基于EntityFramework应用程序的DbContext的生命周期的文章。实际上，[这儿](https://lostechies.com/jimmybogard/2013/12/20/proper-sessiondbcontext-lifecycle-management/)[已经](http://msdn.microsoft.com/en-gb/data/jj729737.aspx)[有非常多](http://blog.longle.net/2013/05/11/genericizing-the-unit-of-work-pattern-repository-pattern-with-entity-framework-in-mvc/)[的讨论](http://blog.longle.net/2013/07/30/bounded-dbcontext-with-generic-unit-of-work-generic-repositories-entity-framework-6-unity-3-0-in-mvc-4/)[这个话题](http://www.davepaquette.com/archive/2013/03/27/managing-entity-framework-dbcontext-lifetime-in-asp-net-mvc.aspx)[的文章](http://coding.abel.nu/2012/10/make-the-dbcontext-ambient-with-unitofworkscope/)[了](http://jupaol.blogspot.co.uk/2012/10/entityframework-context-per-request.html)。

对于许多应用程序，上面这些文章呈现的解决方案（基本上都是利用DI容器对每一个Web请求注入一个DbContext的实例）能够很好的工作。它们的优点也是非常简单——至少第一眼看上去是这样的。

然而，对于某些特定的应用程序，这些解决方案天生的缺陷产生了问题。那就是一些功能变得无法实现或者需要求助于增加复杂的结构或增加丑陋的算法来处理DbContext的创建和管理。

下面是一些真实世界的应用程序的示例，它们促使我重新思考管理DbContext的方式：

   ●一个应用程序包含 ASP.NET MVC和WebAPI构建的Web应用程序。它也可能包含Console和Windows Services构建的后台服务，包含任务调度服务以及那些处理来自于MSMQ和RabbitMQ的消息的服务。我在上面链接的大部分文章都假定所有服务运行在Web请求的上下文里面，但这里涉及的情况显然不是这样。

   ●它从多个数据库读写数据，这些数据库包括一个主数据库，一个从数据库，一个报表数据库和一个日志数据库。它的领域模型被分为几个独立的组，每一个组有自己的DbContext类型，假定只有一个DbContext类型在这儿无论如何也是行不通的。

   ●它非常依赖第三方的远程API,比如Facebook,Twitter或者LinkedIn的API，然而这些API并不支持事务。许多用户操作要求在返回结果给用户之前要先进行几个远程API调用。我在上面链接的大部分文章都假定“一个Web请求只包含一个业务事务”，它应该要么被提交要么被回滚，很显然这条规则在这儿不适用。因为一个远程API调用失败并不意味着我们能神奇的“回滚”之前任何一个已经完成的远程API调用的结果。（例如：当你使用Facebook API向Facebook提交一个状态更新时，你不能因为后续操作失败而回滚向Facebook的状态更新）。因此，在这类应用程序中，一个用户操作经常要求我们执行多个业务事务，每个事务都能独立的持久化。（你可能会争辩说也许有某种方式去重新设计整个系统以避免我们遇到这种情况——当然这是可能的。但如果程序原本就设计成那样，并且运行得很好而且我们不得不处理这种情况呢？）

   ●许多服务都严重并行化，要么借助于异步IO或者（更常见地）通过TPL库提供Task.Run()或者Parallel.Invoke()方法将任务分发到多个线程上去执行。在这种场景下，管理DbContext的大部分常见方法都不管用了。

在这篇文章中，我将深入介绍关于DbContext生命周期管理的各个部分。我们将看到解决这个问题的几种常见策略的优缺点。最后，我们将总结出一个管理DbContext生命周期的策略，它能应对上面提到的各种挑战。

当然，世上没有完美的解决方案。但是在文章的最后，你将拥有为你特定的应用程序做出明智决定的工具和知识。

这是一篇非常长而且详细的文章，它需要一定的时间去阅读和消化。对于基于EntityFramework的应用程序，你选择用来管理DbContext生命周期的策略将是一个重要的决定——它将对你的程序的正确性、可维护性、扩展

性产生重大影响。因此，这值得我们多花一些时间认真考虑后再做出选择。

**本文中要用到的术语**

在这篇文章中，我将多次提到“服务”这个词，它不是指的远程服务（REST或者其它），相反，它是指服务对象——也就是你经常放置业务逻辑实现的地方——这些对象负责执行业务规则和定义业务事务边界。

当然，你的代码基可能用的不同的名字，这取决于你创建应用程序架构所使用的设计模式。因此，我所说的“服务”对你来说也可能叫做“工作流（workflow）”，“协调器（orchestrator）”，“执行者（executor）”，“interactor”，“命令（command）”，“处理者（handler）”或者其它一些名字。

更不用说还有很多应用程序根本就没有定义一个合适的地方来存放业务逻辑，而是随便放在一个地方，比如说MVC应用程序里面的控制器（controllers）。

但是这些都和我们讨论的话题无关——当我说“服务”的时候，就是指存放业务逻辑的地方，它可以是一个随便的控制器（controller）方法或者是一个分层架构中的服务类。

**考虑的关键点**

 当制定或者评估一个管理DbContext生命周期策略的时候，记住它要支持的关键场景和功能是非常重要的。

 下面是一些我认为对大部分程序都很重要的东西。

**你的服务必须控制业务服务的边界（但不是必需控制DbContext的生命周期）**

可能管理DbContext的主要难点是理解DbContext的生命周期与业务事务的生命周期这二者之间的差异和关联。

 DbContext实现了[工作单元模式](http://martinfowler.com/eaaCatalog/unitOfWork.html)：

*维护受业务影响的对象列表，并协调变化和并发问题的解决。*

 在实践中，当你用DbContext实例去加载，更新，添加和删除可持久化的实体时，它会在内存中跟踪这些变化。除非你调用SaveChanges()方法，否则它不会将这些变化持久化到底层数据库。

 一个服务方法，就像上面定义的，它将负责定义业务事务的边界。

 这样的结果就是:

    ●一个服务方法必须用同一个DbContext实例贯穿整个业务事务，这样才能跟踪对可持久化对象的所有修改，并且将这些修改以原子的方式要么提交到底层数据库要么回滚。

    ●你的服务必须是系统中唯一负责调用DbContext.SaveChanges()方法的组件。如果系统的其它模块（比如仓储（repsitory）方法)调用SaveChanges()方法会产生什么结果呢？它将导致提交部分变化，使你的数据处于一种不一致的状态。

    ●SaveChanges()方法必需是在每个业务事务的最后刚好调用一次。如果在业务事务的中间调用也可能会导致不一致的，部分提交的状态。

 一个DbContext实例是可以跨越多个（顺序的）业务事务的。一旦一个业务事务已经完成并且调用DbContext.SaveChanges()方法持久化了所有的修改，那么我们在下一个业务事务中重用同一个DbContext是完全可能的。

 也就是说，DbContext实例的生命周期没有必要和一个单独的业务事务生命周期绑定在一起。

**独立于业务事务的生命周期来管理DbContext实例的生命周期的优缺点**

示例

独立于业务事务生命周期来管理DbContext实例的生命周期的一个常见场景就是Web应用。常见的处理方式是：DbContext实例在每一个请求开始的时候就被创建，然后在这个Web请求的执行过程中被所有的服务调用，并且在请求结束时被释放掉。

优点

下面是关于你为什么要将DbContex实例的生命周管理从业务事务生命周期管理分离开的两个重要原因。

     ●潜在的性能提升。每一个DbContext实例都维护了一个从数据库加载的对象的一级缓存。当你通过主键查询一个实体时，DbContext将优先从一级缓存获取它，在获取不到时，才会尝试从数据库查询。取决于你的数据查询模式，在多个顺序的业务事务中重用同一个DbContext将会因为一级缓存而导致更少的数据库查询。

     ●更多使用延迟加载的场景。如果服务返回可持久化对象（而不是view models或者其它DTOs）那么你就可以利用这些对象的延迟加载功能，加载这些实体的DbContext实例的生命周期必须超越业务事务的范围。如果服务在返回实体对象之前就释放掉了DbContext实例，那么在这些返回对象上触发的任何延迟加载属性都将失败（是否使用延迟加载功能是另一种争论，本文不做深入讨论）。在我们的Web应用示例中，延迟加载常用于服务层返回到控制器动作方法（controller action method)的实体上。这种情况下，服务方法用来加载实体的DbContext实例的生命周期将在整个Web请求过程中（或者至少持续到动作方法的结束）保持激化状态。

**保持DbContext超越业务事务范围都处于激活状态带来的问题**

虽然跨越多个业务事务重用同一个DbContext是可以的，但是DbContext的生命周期还是应该保持得短一些。它的一级缓存最终会过时，并导致并发问题。如果你的应用程序使用[乐观并发](http://msdn.microsoft.com/en-ie/data/jj592904.aspx)策略的话，那么业务事务将会因为DbUpdateConcurrencyException而失败。在Web应用中使用“一个web请求一个DbContext实例”这种策略通常是可以的——因为Web请求时间通常很短。但是在桌面应用中，经常被建议使用的策略是使用“一个窗口（form)一个DbContext实例”，但这会经常出现问题——因此在采纳前应多考虑。

需要注意的是如果你用[悲观并发](http://blogs.msdn.com/b/marcelolr/archive/2010/07/16/optimistic-and-pessimistic-concurrency-a-simple-explanation.aspx)策略的话，那么你不能跨越多个业务事务重用同一个DbContext实例。正确地实现悲观并发策略牵涉到在整个DbContext实例的生命周期中都要以正确的数据库隔离级别保持一个激活的数据库事务——这将防止你在独立的业务事务中独立的提交或者回滚数据。

在超过一个业务事务中重用同一个DbContext实例也可能会导致灾难性的bug——服务方法可能意外的提交了来自上一个失败的业务事务的修改。

最后，在你的服务方法的外面来管理DbContext实例的生命周期会倾向于把你的应用程序和一个制定的基础架构绑定在一起——从长远来看，这使你的应用程序更加不灵活并且更难演进和维护。

例如，对于一个刚开始很简单的Web应用程序来说，它依赖于“一个Web请求创建一个DbContext实例”的策略来管理DbContext的生命周期时，这很容易掉进一个圈套——在控制器（controller)或者视图(view)中使用延迟加载功能或者在服务方法之间传递可持久化对象——假定这些场景都会使用同一个DbContext实例。当不可避免地要引入多线程或者转移这些操作到后台WindowsService去的时候，这些精心设计的沙堡就崩塌了——因为这儿没有更多的Web请求来绑定DbContext实例了。

因此，建议避免独立于业务事务来管理DbContext实例的生命周期。应当在每一个服务方法内部创建它们自己的DbContext实例，并在业务事务结束的时候释放该实例。

这将防止在服务外面使用延迟加载（也可以让服务方法返回传输对象而非可持久化对象来防止在服务外面使用延迟加载）。另外，最好不要传递可持久化对象给服务——因为这些对象没有依附在服务将要使用的DbContext上。虽然有这么多限制，但从长远来看，它将给我们带来很好的灵活性和可维护性。

**你的服务必须控制数据库事务的范围和隔离级别**

如果你的应用程序使用的数据库提供的事务支持ACID四个要素（如果你用的是EntityFramework,那么肯定就是了），那么你的服务控制数据库事务的范围和隔离级别就很有必要了，否则你就不能写出正确的代码。

我们将在后面看到，EntityFramework将所有的写操作用一个显示数据库事务打包在一起——默认情况下，将用[READ COMMITTED](http://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_committed) 隔离级别——也就是SQL Server的默认设置——这能适合大部分业务事务。尤其是你依赖于[乐观并发](http://en.wikipedia.org/wiki/Optimistic_concurrency_control)策略去检测和避免”更新冲突“的情况，更是如此。

无论如何，大部分应用程序仍然需要为某些特定的操作使用其它的隔离级别。

比如，执行报表查询的场景，你可能会觉得脏读不是问题从而选择使用[READ UNCOMMITTED](http://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_uncommitted)隔离级别——这样可以消除锁竞争。

并且有的业务规则可能要去使用REPEATABLE  READ 或者 SERIALIZABLE 隔离级别（尤其是你的项目使用悲观并发策略的话）。这些场景就需要服务显示控制事务范围了。

**管理DbContext的方式应当独立于系统架构**

软件系统架构和设计模式会随着时间进化以适应新的业务规则和负载升级。

你肯定不想因为选择的管理DbContext生命周期的策略绑定在一个特定的架构而限制你对其进行升级。

**管理DbContext的方式应当独立于应用程序类型**

虽然现在大部分应用程序都开始于一个Web应用程序，但是你选择用来管理DbContext生命周期的策略不应当假定你的服务方法只会在基于Web请求的上下文中被调用。一般来说，你的服务层（如果有的话）应当独立于使用它的应用程序的类型。

在应用程序启动不久后，你就可能需要创建命令行工具去执行运维任务或者创建Windows服务来处理定时任务或者需要长时间运行的后台操作。当这种情况发生的时候，你期望能够为你的控制台应用程序或者Windows服务程序引用你的服务所在的程序集。你肯定不愿看到需要完全重构管理DbContext实例的方式后你的服务才能被不通的应用程序类型使用。

**管理DbContext的方式应当支持多种DbContext派生类**

如果你的应用程序需要访问多个数据库（比如报表数据库，日志/审计数据库）或者你将你的领域模型分离成多个聚合组，那么你就将有多个DbContext派生类。

对于NHibernate用户来说，这就相当于管理多个SessionFactory实例。

无论你选择哪种策略都应当能让服务选择需要的DbContext类型。

**管理DbContext的方式应当能支持EF6提供的异步工作流**

在.NET 4.5中，ADO.NET引入了支持[异步数据库查询](http://blogs.msdn.com/b/adonet/archive/2012/04/20/using-sqldatareader-s-new-async-methods-in-net-4-5-beta.aspx)的功能。随后异步支持也被包括在EntityFramework6中——它允许你使用一个完整的异步工作流来读写数据。

无需多说，无论你选择什么来管理DbContext,但它必须能很好地与EF异步功能协调工作。

**DbContext的默认行为**

通常来说，DbContext的默认行为可以被描述为：“默认情况下就能做正确的事”。

下面是你应该记在脑海里面的几个关于EntityFramework的重要行为。这个列表描述了EF访问SqlServer的行为。用其它的数据库可能会略有差异。

**DbContext不是线程安全的**

你千万不要从多个线程同时去访问DbContext派生类实例。这可能导致将多个查询通过一个相同的数据库连接被同时发送了出去——它将破坏DbContext维护的一级缓存的状态——它们被用来提供标识映射（Identity Map),变更追踪和工作单元的功能。

在一个多线程应用程序中，你必须为每一个线程创建一个独立的DbContext派生类实例。

问题来了，如果DbContext不是线程安全的，那么它怎么支持EF6的异步功能呢？其实很简单：只需要保证在任何时刻只有一个异步操作被执行（就像[EF的支持异步模式的规范](https://entityframework.codeplex.com/wikipage?title=Task-based%20Asynchronous%20Pattern%20support%20in%20EF.#ThreadSafety)描述的那样）。如果你尝试在同一个DbContext实例上并发的执行多个操作，比如通过DbSet<T>.ToListAsync()方法并发地执行多个查询语句，你将会得到一个带有下面消息的NotSupportedException。

*A second operation started on this context before a previous asynchronous operation completed. Use 'await' to ensure that any asynchronous operations have completed before calling another method on this context. Any instance members are not guaranteed to be thread safe.*

EF的异步功能是为了支持异步编程模型，而不是并发编程模型。

**当且仅当SaveChanges()方法被调用的时候，修改才会被持久化**

任何对实体的修改，包括更新，插入或者删除，当且仅当DbContext.SaveChanges()被调用的时候才会被持久化到数据库。如果DbContext实例在SaveChanges()方法被调用之前就被释放掉了，那么这些更新操作，插入操作，删除操作没有一条能持久化到底层数据库。

下面是用EF来实现一个业务事务的规范方式：

using (var context = new MyDbContext(ConnectionString))

{

/\*

\* 业务逻辑放在这儿. 通过context添加，修改，删除数据。

\*

\* 抛出任何异常就可以回滚所有变化。

\*

\* 直到业务事务完成，否则不能调用SaveChanges()方法

\* 也就是说不能部分或者中间保存。

\* 每一个业务事务只能刚好调用一次SaveChanges()方法 。

\*

\* 如果你发现你自己需要在一个业务事务里面多次调用

\* SaveChanges()方法，那就意味着你在一个服务方法

\* 里面实现多个业务事务。这绝对是灾难的“必备良药”。

\* 调用你的服务的客户端会很自然的假定你的服务方法

\* 以原子的行为提交或者回滚——但你的服务却可能

\* 部分提交，让系统处于一个不一致的状态。

\*

\* 在这种情况下，将你的服务方法重构成多个服务方法——

\* 每一个服务方法刚好实现了刚好一个业务事务。

\*/

[...]

// 完成业务事务并且持久化所有变化 。

context.SaveChanges();

// 在这行代码之后变化不可能回滚了。

// context.SaveChanges()应当是任何业务事务

// 的最后一行代码。

}

[复制代码](javascript:void(0);)

**NHibernate用户注意事项**

如果你拥有NHibernate背景，那么可以告诉你的是EF将变化持久化到数据库的方式是它与NHibernate的最大不同。

在NHibernate中,Session操作默认情况下处于[AutoFlush模式](http://www.nhforge.org/doc/nh/en/#manipulatingdata-flushing)。在这种模式下，Session将在执行任何‘select’操作之前自动将所有变化持久化到数据库——确保已持久化到数据库的实体和它们在Session中的内存状态保持一致。对NHibernate来说，EF的默认行为相当于将Session.FlushMode设置为Never。

EF的这个行为可能会导致一些微妙的bug——查询意外的返回过时的或者不正确的数据。默认情况下NHibernate是绝不可能出现这种情况的。但从另外一方面来说，这却又极大的简化了数据库事务管理的问题。

在NHibernate中最棘手的问题之一就是正确的管理数据库事务。由于NHibernate的Session可以在它的整个生命周期中的任何时间点自动地将未持久化的变化持久化到数据库，并且可能在一个业务事务里面持久化多次——这儿没有一个定义良好的点或者方法来开启数据库事务以确保所有的修改以原子的行为提交或者回滚。

在NHibernate中正确管理数据库事务的唯一可靠方法就是将你的所有服务方法打包在一个显式数据库事务中。这就是大部分基于NHibernate的应用程序的处理方式。

这种方式的负面效应就是它要求打开一个数据库连接和事务的时间比实际需要的要更长——因此增加了数据库锁的竞争和数据库死锁发生的可能性。开发者也很容易不经意的执行一个长时间计算或者一个远程服务方法的调用而没有意识到甚至根本就不知道他们是在一个数据库事务打开的上下文中。

EF的方式——只有SaveChanges()方法必须被打包在一个显式数据库事务中（当然使用一个REPEATABLE READ 或者SERIALIZABLE隔离级别的情况例外），保证了数据库连接和事务保持尽可能的短暂。

**使用自动提交事务（AutoCommit transaction)来执行读取操作**

DbContext不支持打开一个显式事务来执行读取操作。它依赖于SQL Server的自动提交事务（[Autocommit Transaction](http://technet.microsoft.com/en-us/library/ms187878(v=sql.105).aspx)) (或者 [隐式事务](http://technet.microsoft.com/en-us/library/ms188317(v=sql.105).aspx)(Implicit Transaction)——如果你启用了它们的话，但那相对来说不是常见的操作）。自动提交事务（或者隐式事务）将会使用数据库引擎被配置的默认事务隔离级别（对SQL Server来说就是READ COMMITTED）。

如果你已经工作有一段时间，尤其是如果你以前使用过NHibernate，那么你可能听说过“[自动提交事务（或者隐式事务）是糟糕的](http://www.hibernatingrhinos.com/products/nhprof/learn/alert/donotuseimplicittransactions)”。实际上，依赖于自动提交事务的写操作可能在性能上产生[灾难性影响](http://dba.stackexchange.com/a/43256)。

但对于读操作来说情况就大不一样了。你可以跑下面的SQL脚本亲自去看看。对select语句来说，自动提交事务或者隐式事务都不会有任何明显的性能影响。

[复制代码](javascript:void(0);)

/\*

\* 用自动提交事务，隐式事务，显式事务分部执行10000

\* 次select查询.

\*

\* 这些脚本假定数据库包含一张Users表，它有一个列名为Id

\* 类型为INT的列。

\*

\* 如果你在SQL Server Management Studio里面运行的话

\* 右键查询窗口，进入查询选项 -> 点击结果并勾选

\* “执行后放弃结果”。否则你的测试结果将会被网格的

\* 刷新验证影响

\*/

---------------------------------------------------

-- 自动提交事务

-- 6 秒

DECLARE @i INT

SET @i = **0**

WHILE @i < **100000**

BEGIN

SELECT Id

FROM dbo.Users

WHERE Id = @i

SET @i = @i + **1**

END

---------------------------------------------------

-- 隐式提交事务

-- 6 秒

SET IMPLICIT\_TRANSACTIONS ON

DECLARE @i INT

SET @i = **0**

WHILE @i < **100000**

BEGIN

SELECT Id

FROM dbo.Users

WHERE Id = @i

SET @i = @i + **1**

END

COMMIT;

SET IMPLICIT\_TRANSACTIONS OFF

----------------------------------------------------

-- 显示事务

-- 6 秒

DECLARE @i INT

SET @i = **0**

BEGIN TRAN

WHILE @i < **100000**

BEGIN

SELECT Id

FROM dbo.Users

WHERE Id = @i

SET @i = @i + **1**

END

COMMIT TRAN

很显然，如果你需要用一个比默认READ COMMITTED更高的隔离级别的话，那么所有读操作都将是显式数据库事务的一部分。在那种情况下，你需要自己开启事务——EF将不会为你做这个。但这通常只会为指定的业务事务做特别处理。EF的默认设置能适合大部分业务事务。

**使用显式事务来执行写操作**

EF通过DbContext.SaveChanges()方法自动地将所有操作打包在一个显式数据库事务里面——以确保应用在context的所有修改要么完全提交要么完全回滚。

EF写操作使用数据库引擎配置的默认事务隔离级别（对SQL Server来说就是READ COMMITTED）。

**NHibernate用户注意事项**

这是EF和NHibernate之间的另一个很大的不同点。在NHibernate中，数据库事务完全掌握在开发者手中。NHibernate的Session永远不会自动地打开一个显式数据库事务。

**你可以重写EF的默认行为并控制数据库事务范围和隔离级别**

[复制代码](javascript:void(0);)

using (var context = new MyDbContext(ConnectionString))

{

using (var transaction =context.BeginTransaction(IsolationLevel.RepeatableRead))

{

[...]

context.SaveChanges();

transaction.Commit();

}

}

[复制代码](javascript:void(0);)

手动控制数据库事务范围的一个非常明显的副作用就是你必须在整个事务范围中让数据库连接和事务保持打开。

你应当尽可能的让这个事务范围生命周期短暂。打开一个数据库事务运行太长时间可能会对应用程序的性能和可扩展性有非常巨大的影响。特别指出的是，尽量不要再一个显示事务范围内调用其它的服务方法——它们可能执行长时间运行的操作而没有意识到它们是在一个打开的数据库事务内被调用。

**EF没有内建的方式来重写用作自动提交事务和自动显式事务的默认隔离级别**

就像上面提到的，EF依赖自动提交事务来执行读操作并且当调用SaveChanges()方法的时候自动以数据库配置的默认隔离级别开启一个显式事务。

很不幸的是没有内建的方式来重写这些隔离级别，如果你想用另一个隔离级别，你必须自己开启和管理数据库事务。

**通过DbContext打开的数据库连接自动加入一个周围环境的TransactionScope**

另外，你也可以用TransactionScope来控制事务范围和隔离级别。EF打开的数据库连接自动加入周围环境的TransactionScope。

在EF6之前，使用TransactionScope是唯一可靠的方式来控制数据库事务范围和隔离级别。

在实践中，除非你真的需要一个分布式事务，否则尽量避免使用TransactionScope。TransactionScope，通常指分布式事务，对大部分应用程序来说都是不必要的。并且它们通常会带来比它们解决的问题都要更多的问题。如果你真的需要一个分布式事务的话，可以查看EF文档章节——[在EF中使用TransactionScope](http://msdn.microsoft.com/en-gb/data/dn456843.aspx#transactionScope)。

**DbContext实例应当被释放掉（但是如果没有释放掉，也可能没事）**

DbContext实现了IDisposable接口，因此一旦它们不需要了就应当尽快释放。

然而在实践中，除非你选择显式控制DbContext使用的数据库连接或者事务，否则不调用DbContext.Dispose()方法也不会引起任何问题——[就像Diego Vega，一个EF团队成员解释的那样](http://blog.jongallant.com/2012/10/do-i-have-to-call-dispose-on-dbcontext.html#.U9efX_ldXpo)。

这是一个好消息——因为你会发现很多代码不能正确地释放DbContext实例。尤其是那些尝试用DI容器来管理DbContext实例生命周期的情况——实际情况比听起来要棘手得多。

一个DI容器，比如说StructureMap，它不支持释放它创建的组件。因此，如果你依赖StructureMap来创建DbContext实例，那么它们将不会被释放掉——不管你为它们设置的什么生命周期方式。使用像这样的DI容器来管理可释放组件的唯一正确方式就是复杂你的DI配置并且使用一个嵌套依赖注入容器——就像[Jeremy Miller描述的那样](http://codebetter.com/jeremymiller/2010/01/06/how-dovetail-uses-structuremap-with-nhibernate/)。

# 环境上下文DbContext vs 显式DbContext vs 注入DbContext

在任何基于EF项目之初的一个关键决定就是你的代码如何传递DbContext实例到下面真正访问数据库的方法里面。

就像我们在上面看到的，创建和释放DbContext的职责属于顶层服务方法。数据访问代码，就是那些真正使用DbContext实例的代码，可能经常在一个独立的部分里面——可能深入在服务实现类的一个私有方法里面，也可能在一个查询对象里面或者一个独立的仓储层里面。

顶层服务方法创建的DbContext实例需要找到一个传递到这些方法的方式。

这儿有三个想法来让数据访问代码访问DbContext:环境上下文DbContext,显式DbContext或者注入DbContext。每一种方式都有它们各自的优缺点，让我们来逐个分析。

## 显式DbContext

它看起来是怎么样的

使用显式DbContext方法，顶层服务创建一个DbContext实例然后通过一个方法的参数传递至数据访问的部分。在一个传统的包含服务层和仓储层的三层架构中，大概看起来就是这样：

public class UserService : IUserService

{

private readonly IUserRepository \_userRepository;

public UserService(IUserRepository userRepository)

{

if (userRepository == null) throw new ArgumentNullException("userRepository");

\_userRepository = userRepository;

}

public void MarkUserAsPremium(Guid userId)

{

using (var context = new MyDbContext())

{

var user = \_userRepository.Get(context, userId);

user.IsPremiumUser = true;

context.SaveChanges();

}

}

}

public class UserRepository : IUserRepository

{

public User Get(MyDbContext context, Guid userId)

{

return context.Set<User>().Find(userId);

}

}

（在这个故意为之的示例里面，仓储层的作用当然是完全无意义的。在一个真实的应用程序中，你将期望仓储层更加饱满。另外，如果你真的不想让你的服务直接依赖EF,你可以抽象你的DbContext为“IDbContext”之类的并且通过一个抽象工厂来创建它。）

**优点**

这种方式是到目前为止而且永远也是最简单的方式。它使得代码非常简单易懂而且易于维护——即使对于那些对代码不是很熟悉的开发人员来说也是这样的。

这儿没有任何神奇的地方。DbContext实例不会凭空创建。它是在一个清晰的明显的地方被创建——如果你好奇DbContext来源于哪儿的话你也可以通过调用栈非常容易的找到。

**缺点**

这种方式最主要的缺点是它要求你去污染所有你的所有仓储方法（如果你有一个仓储层的话），同样你的大多服务方法也会有一个强制的DbContext参数（或者某种类型的IDbContext抽象——如果你不希望绑定到具体实现的话——但是问题仍然存在）。所以你可能会看到某些方法注入模式的应用。

你的仓储层方法要求提供一个显式的DbContext作为参数也不是什么大问题。实际上，它甚至可以看着已经好事——因为他消除了潜在的歧义——就是这些查询究竟用的哪一个DbContext。

但是对于服务层情况就大不一样了。因为大部分你的服务方法都不会用DbContext,尤其是你将数据访问代码隔离在一个查询对象或者仓储层里面的时候。因此，这些服务方法提供了一个DbContext参数的目的仅仅是为了将他们传递到下层真正需要用到DbContext的方法里面。

这很容易变得十分丑陋。尤其是你的应用程序需要使用多个DbContext的时候，将导致你的服务方法要求两个甚至更多的DbContext参数。这将混淆你的方法的契约——因为你的服务方法现在强制要求一个它们既不需要也不会用而仅仅是为了满足底层方法依赖的参数。

[Jon Skeet写了一篇关于显式DbContext vs 隐式DbContext的文章，但没有提供一个好的解决方案](http://blogs.msmvps.com/jonskeet/2010/11/08/the-importance-of-context-and-a-question-of-explicitness/)。

然而，这种方法的超级简单性还是很难被其它方法打败的。

**环境上下文DbContext**

它看起来是怎么样的

NHibernate用户应当是对这种方式非常熟悉——因为[环境上下文模式（ambient context pattern)](http://blogs.msdn.com/b/ploeh/archive/2007/07/23/ambientcontext.aspx)是在NHibernate世界里面管理NHibernate的Session（它相当于EF的DbContext）的主要方式。NHibernate甚至对该模式有内置支持，叫做[上下文session(contextual session)](http://www.nhforge.org/doc/nh/en/#architecture-current-session)。

在.NET自身，这种模式也是用得相当广泛。你可能已经用过HttpContext.Current或者TransactionScope，两者都是依赖于环境上下文模式。

使用这种模式，顶层服务方法不仅创建用于当前业务事务的DbContext，而且还要将其注册为环境上下文DbContext。然后数据访问代码就可以在需要时候获取这个环境上下文DbContext了。再也不需要传递DbContext实例。

Anders Abel已经写过一篇文章——[简单实现环境上下文DbContext](http://coding.abel.nu/2012/10/make-the-dbcontext-ambient-with-unitofworkscope/)——它依赖ThreadStatic变量来存储DbContext。去看看吧——它比听起来都还要更简单。

**优点**

这种方式的优点是显然的。你的服务和仓储方法现在对DbContext参数已经自由了（也就是说服务和仓储方法不需要DbContext作为参数了）——让你的接口更加干净并且你的方法契约更加清晰——因为它们现在只需要获取他们真正需要使用的参数了。再也不需要遍地传递DbContext实例了。

**缺点**

无论如何这种方式引入了一定程度的魔法——它让代码更难理解和维护。当看到数据访问代码的时候，不一定容易发现环境上下文DbContext来自于哪儿。你不得不希望在调用数据访问代码之前某人已经将它注册了。

如果你的应用程序使用多个DbContext派生类，比如，如果你连接多个数据库或者如果你将领域模型划分为几个独立的组，那么对于顶层服务来说就很难知道应当创建和注册哪些DbContext了。使用显式DbContext,数据访问方法要求提供它们需要的DbContext作为参数，因此就不存在歧义的可能。但是使用环境上下文方式，顶层服务方法必须知道下层数据访问代码需要哪种DbContext类型。我们将在后面看到一些解决这个问题的十分干净的方式。

最后，我在上面链接的环境上下文DbContext例子只能在单线程模型很好的工作。如果你打算使用[EF的异步查询功能](http://msdn.microsoft.com/en-ie/data/jj819165.aspx)的话，它就不能工作了。在一个异步操作完成后，你很可能发现你自己已经在另外一个线程——不再是之前创建DbContext的线程。在许多情况下，它意味着你的环境上下文DbContext将消失。这个问题可以解决，但是它要求一些深入的理解——在.NET世界里面如何多线程编程，TPL和异步工作背后的原理。我们将在文章最后看到这些。

**注入DbContext**

**它看起来是怎么样的**

最后一种比较重要的方式，注入DbContext方式经常被各种文章和博客提及用来解决DbContext生命周期的问题。

使用这种方式，你可以让DI容器管理DbContext的生命周期并且在任何组件（比如仓储对象）需要的时候就注入它。

看起来就是这样的：

[复制代码](javascript:void(0);)

public class UserService : IUserService

{

private readonly IUserRepository \_userRepository;

public UserService(IUserRepository userRepository)

{

if (userRepository == null) throw new ArgumentNullException("userRepository");

\_userRepository = userRepository;

}

public void MarkUserAsPremium(Guid userId)

{

var user = \_userRepository.Get(context, userId);

user.IsPremiumUser = true;

}

}

public class UserRepository : IUserRepository

{

private readonly MyDbContext \_context;

public UserRepository(MyDbContext context)

{

if (context == null) throw new ArgumentNullException("context");

\_context = context;

}

public User Get(Guid userId)

{

return \_context.Set<User>().Find(userId);

}

}

[复制代码](javascript:void(0);)

然后你需要配置你的DI容器以使用合适的生命周期策略来创建DbContext实例。你将发现一个常见的建议是对于Web应用程序使用一个PerWebRequest生命周期策略，对于桌面应用使用PerForm生命周期策略。

**优点**

好处与环境上下文DbContext策略相似：代码不需要到处传递DbContext实例。这种方式甚至更进一步：在服务方法里面根本看不到DbContext。服务方法完全不知道EF的存在——第一眼看起来可能很好，但很快就会发现这种策略会导致很多问题。

**缺点**

不管这种策略有多流行，它确切是有非常重大的缺陷和限制。在采纳之前先了解它是非常重要的。

**太多魔法**

这种方式的第一个问题就是太依赖魔法。当需要保证你的数据——你最珍贵的资产的正确性和一致性的时候，“魔法”不是你想听到太频繁的一个词。

这些DbContext实例来自于哪里？业务事务的边界如何定义和在哪儿定义？如果一个服务方法依赖两个不同的仓储类，那么这两个仓储式都访问同一个DbContext实例呢还是它们各自拥有自己的DbContext实例？

如果你是一个后端开发人员并且在开发基于EF的项目，那么想要写出正确代码的话，就必须知道这些问题的答案。

答案并不明显，它需要你详细查看DI容器的配置代码才能发现。就像我们前面看到的，要正确设置这些配置不是第一眼看上去那么容易，相反，它可能是非常复杂而且容易出错的。

**不清晰的业务事务边界**

可能上面示例代码最大的问题是：谁负责提交修改到数据库？也就是谁调用DbContext.SaveChanges()方法？它是不清晰的。

你可以仅仅是为了调用SaveChanges()方法而将DbContext注入你的服务方法。那将是更令人费解和容易出错的代码。为什么服务方法在一个既不是它创建的又不是它要使用的DbContext对象上调用SaveChanges()方法呢？它将保存什么修改？

另外，你也可以在你的所有仓储对象上定义一个SaveChanges()方法——它仅仅委托给底层的DbContext。然后服务方法在仓储对象上调用SaveChanges()方法。这也将是非常具有误导性的代码——因为他暗示着每一个仓储对象实现了它们自己的工作单元并且可以独立于其它仓储对象持久化自己的修改——这显然不是正确的，因为他们实际上是用的都是同一个DbContext实例。

有些时候你会看到还有一种方式：让DI容器在释放DbContext实例之前调用SaveChanges()方法。这是一个灾难的方法——值得一篇文章来描述。

简短来说，DI容器是一种基础设施组件——它对它管理的组件的业务逻辑一无所知。相反，DbContext.SaveChanges()方法定义了一个业务事务的边界——也就是说它是以业务逻辑为中心的。混合两种完全不相关的概念在一起将会引起很多问题。

话虽如此，如果你知道“[仓储层已死（Repository is Dead）](http://ayende.com/blog/3955/repository-is-the-new-singleton)”运动。谁来调用DbContext.SaveChanges()方法根本不是问题——因为你的服务方法将直接使用DbContext实例。它们因此很自然的成为调用SaveChanges()方法的地方。

当你使用注入DbContext策略的时候，不管你的应用程序的架构模式，你将还会遇到一些其它的问题。

**强制你的服务变成有状态的**

一个值得注意的地方是DbContext不是一个服务。它是一个资源，一个需要释放的资源。通过将它注入到你的数据访问层。你将使那一层的所有上层——很可能就是整个应用程序，都变成有状态的。

这当然不是世界末日但它却肯定会让DI容器的配置变得更复杂。使用无状态的服务将提供巨大的灵活性并且使得配置它们的生命周期变得不会出错。一旦你引入状态化的服务，你就得认真考虑你服务的生命周期了。

注入DbContext这种方式在项目刚开始的时候很容易使用（PerWebRequest或者Transient生命周期都能很好的适应简单的web应用），但是控制台应用程序，Window服务等让它变得越来越复杂了。

**阻止多线程**

另外一个问题（相对前一个来说）将不可避免的狠咬你一口——注入DbContext将阻止你在服务中引入多线程或者某种并行执行流的机制。

请记住DbContext(就像NHibernate中的Session)不是线程安全的。如果你需要在一个服务中并行执行多个任务，你必须确保每个任务都使用它们自身的DbContext实例，否则应用程序将在运行时候崩溃。但这对于注入DbContext的方式来说是不可能的事情因为服务不能控制DbContext实例的创建。

你怎么修复这个缺陷呢？答案是不容易。

你的第一直觉可能是将你的服务方法修改为依赖DbContext工厂而非直接依赖DbContext。这将允许它们在需要的时候创建它们自己的DbContext实例。但这样做将会有效地推翻注入DbContext这种观点。如果你的服务通过一个工厂创建它们自己的DbContext实例，这些实例再也不能被注入了。那将意味着服务将显式传递这些DbContext实例到下层需要它们的地方（比如说仓储层）。这样你又回到了之前我们讨论的显式DbContext策略了。我可以想到一些解决这些问题的方法——但所有这些方法感觉起来像不寻常手段而不是干净并且优雅的解决方案。

另外一种解决这个问题的方式就是添加更多复杂的层，引入一个像RabbitMQ 的中间件并且让它为你分发任务。这可能行得通但也有可能行不通——完全取决于为什么你需要引入并发性。但是在任何情况下，你可能都不需要也不想要附加的开销和复杂性。

使用注入DbContext的方式，你最好限制你自己只使用单线程代码或者至少是一个单一的逻辑执行流。这对于大部分应用程序都是完美的，但是在特定情况下，它将变成一个很大的限制。

**DbContextScope:一个简单的，正确的并且灵活的管理DbContext实例的方式**

应当是来看看一种更好地管理这些DbContext实例方式的时候了。

在下面呈现的方式依赖于DbContextScope，它是一个定制的组件，实现了上面说到的环境上下文DbContext方式。DbContextScope和它依赖的相关类的源代码都[放到了GitHub上面](https://github.com/mehdime/DbContextScope)。

如果你熟悉TransactionScope类，那么你就已经知道如何使用一个DbContextScope了。它们在本质上十分相似——唯一的不同是DbContextScope创建和管理DbContext实例而非数据库事务。但是就像TransactionScope一样，DbContextScope是基于环境上下文的，可以被嵌套，可以有嵌套行为被禁用，也可以很好地与异步工作流协作。

下面是DbContextScope的接口：

[复制代码](javascript:void(0);)

public interface IDbContextScope : IDisposable

{

void SaveChanges();

Task SaveChangesAsync();

void RefreshEntitiesInParentScope(IEnumerable entities);

Task RefreshEntitiesInParentScopeAsync(IEnumerable entities);

IDbContextCollection DbContexts { get; }

}

[复制代码](javascript:void(0);)

DbContextScope的目的是创建和管理在一个代码块内使用的DbContext实例。一个DbContextScope因此有效的定义了一个业务事务的边界。我将在后面解释为什么我没有将其命名为“工作单元（UnitOfWork）”或者“工作单元范围（UnitOfWorkScope）”——它们拥有更广泛的使用场景。

你可以直接实例化一个DbContextScope，你也可以依赖IDbContextScopeFactory——它提供一个方便的方法并使用最常见的配置来创建一个DbContextScope：

[复制代码](javascript:void(0);)

public interface IDbContextScopeFactory

{

IDbContextScope Create(DbContextScopeOption joiningOption = DbContextScopeOption.JoinExisting);

IDbContextReadOnlyScope CreateReadOnly(DbContextScopeOption joiningOption = DbContextScopeOption.JoinExisting);

IDbContextScope CreateWithTransaction(IsolationLevel isolationLevel);

IDbContextReadOnlyScope CreateReadOnlyWithTransaction(IsolationLevel isolationLevel);

IDisposable SuppressAmbientContext();

}

[复制代码](javascript:void(0);)

**典型用法**

使用DbContextScope,你的典型服务方法将看起来是这样的：

[复制代码](javascript:void(0);)

public void MarkUserAsPremium(Guid userId)

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

var user = \_userRepository.Get(userId);

user.IsPremiumUser = true;

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

在一个DbContextScope里面，你可以用两种方式访问scope管理的DbContext实例。你可以像下面这样通过DbContextScope.DbContexts属性获取它们：

[复制代码](javascript:void(0);)

public void SomeServiceMethod(Guid userId)

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

var user = dbContextScope.DbContexts.Get<MyDbContext>.Set<User>.Find(userId);

[...]

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

但那当然也是DbContextScope在方法里面提供的唯一方式。如果你需要在其它地方（比如说仓储类）访问环境上下文DbContext实例，你可以依赖IAmbientDbContextLocator，像下面这样使用：

[复制代码](javascript:void(0);)

public class UserRepository : IUserRepository

{

private readonly IAmbientDbContextLocator \_contextLocator;

public UserRepository(IAmbientDbContextLocator contextLocator)

{

if (contextLocator == null) throw new ArgumentNullException("contextLocator");

\_contextLocator = contextLocator;

}

public User Get(Guid userId)

{

return \_contextLocator.Get<MyDbContext>.Set<User>().Find(userId);

}

}

[复制代码](javascript:void(0);)

这些DbContext实例是延迟创建的并且DbContextScope跟踪它们以确保在它的范围内任何DbContext派生类只会被创建一个实例。

你将注意到服务方法在整个业务事务范围内不需要知道究竟需要哪种DbContext派生类型。它仅仅需要创建一个DbContextScope并且在其范围内的需要访问数据库的任何组件都能获取到它们需要的DbContext。

**嵌套范围(Nesting Scopes)**

一个DbContextScope当然可以被嵌套。让我们假定你已经有一个服务方法，它将用户标记为优质用户，像下面这样：

[复制代码](javascript:void(0);)

public void MarkUserAsPremium(Guid userId)

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

var user = \_userRepository.Get(userId);

user.IsPremiumUser = true;

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

你正在实现一个新的功能，它要求能在一个业务事务内标记一组用户为优质用户。你可以像下面这样很容易的完成它：

[复制代码](javascript:void(0);)

public void MarkGroupOfUsersAsPremium(IEnumerable<Guid> userIds)

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

foreach (var userId in userIds)

{

// 通过MarkUserAsPremium()创建的子范围将加入我们的范围，

// 因此它能重用我们的DbContext实例，并且对SaveChanges()

// 的调用将没有任何作用。

MarkUserAsPremium(userId);

}

// 修改都将只有在这儿才能被保存，在顶层范围内，以确保所有的修改

// 以原子的行为要么提交要么回滚。

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

（当然这是实现这个指定功能的一种非常不高效的方式，但是它说明了如何实现嵌套事务（范围））

这使得创建一个能组合使用多个其它多个服务方法的服务成为可能。

**只读范围(Read-only scopes)**

如果一个服务方法是只读的，那么在方法返回之前必须在DbContextScope上调用SaveChanges()方法将是痛苦的，但是如果不调用也有不妥，因为：

  1.它将使代码审查和维护更困难（你究竟是有意没有调用SaveChanges()还是你忘了调用呢？）

  2.如果你开启一个显式数据库事务（我们将在后面看到如何这样做），不调用SaveChanges()将导致事务被回滚。数据库监控系统将通常认为事务回滚意味着应用程序错误。造成一种假的回滚不是一个好主意。

DbContextReadOnlyScope用来解决这个问题。下面是它的接口：

public interface IDbContextReadOnlyScope : IDisposable

{

IDbContextCollection DbContexts { get; }

}

你可以像下面这样使用它：

[复制代码](javascript:void(0);)

public int NumberPremiumUsers()

{

using (\_dbContextScopeFactory.CreateReadOnly())

{

return \_userRepository.GetNumberOfPremiumUsers();

}

}

[复制代码](javascript:void(0);)

**异步支持**

DbContextScope将如你期望的能很好的在异步执行流中工作：

[复制代码](javascript:void(0);)

public async Task RandomServiceMethodAsync(Guid userId)

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

var user = await \_userRepository.GetAsync(userId);

var orders = await \_orderRepository.GetOrdersForUserAsync(userId);

[...]

await dbContextScope.SaveChangesAsync();

}

}

[复制代码](javascript:void(0);)

在上面的例子中，OrderRepository.GetOrdersForUserAsync()方法将能看到并且访问环境上下文DbContext实例——尽管事实上它是在另一个线程而非DbContextScope最初被创建的线程上被调用。

使这一切成为可能的原因是DbContextScope将它自己存储在CallContext上面的。CallContext通过异步点自动流转。如果你对它背后的工作原理很好奇，Stephen Toub已经写过[一篇关于它的优秀文章](http://blogs.msdn.com/b/pfxteam/archive/2012/06/15/executioncontext-vs-synchronizationcontext.aspx)。但是如果你想要的只是使用DbContextScope，你只需要知道：它就是能工作。

**警告**：当你在异步流中使用DbContextScope的时候，有一件事情你必须记住：就像TransactionScope，DbContextScope仅支持在一个单一的逻辑流中使用。

也就是说，如果你尝试在一个DbContextScope范围内开启多个并行任务（比如说创建多个线程或者多个TPL任务），你将陷入大麻烦。这是因为环境上下文DbContextScope将流转到你并行任务使用的所有线程。如果在这些线程中的代码需要使用数据库，它们就都将使用同一个环境上下文DbContext实例，导致多个线程同时使用同一个DbContext实例。

通常，在一个单独的业务事务中并行访问数据库没有什么好处除了增加复杂性。在业务事务中的任何并行操作都应当不要访问数据库。

无论如何，如果你针对需要在一个DbContextScope里面开启一个并行任务（比如说你要通过业务事务的结果独立的执行一些后台处理），你必须在开启并行任务之前禁用环境上下文DbContextScope，你可以像下面这样简单处理：

[复制代码](javascript:void(0);)

public void RandomServiceMethod()

{

using (var dbContextScope = \_dbContextScopeFactory.Create())

{

// 使用环境上下文context执行一些代码

[...]

using (\_dbContextScopeFactory.SuppressAmbientContext())

{

// 在这儿，开启的并行任务将不能使用环境上下文context.

[...]

}

// 在这儿，环境上下文将再次变为可用。

// 可以像平常一样执行更多的代码

[...]

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

**创建一个非嵌套的DbContextScope**

这是一个我期望大部分应用程序永远不需要用到的高级功能。当使用它的时候要认真对待——因为它能导致一些诡异的问题并且很快导致维护的恶魔。

有些时候，一个服务方法可能需要将变化持久化到底层数据库而不管整个业务事务的结果，就像下面这些情况：

   1.需要在一个全局的地方记录不应当回滚的信息——即使业务事务失败。一个典型的例子就是日志或者审计记录。

   2.它需要记录一个不能回滚的操作的结果。一个典型的例子就是服务方法和非事务性的远程服务或者API交互。例如，如果你的服务方法使用Facebook API提交一个状态更新然后在本地数据库记录新创建的状态。这个记录必须被持久化即使整个业务事务因为在调用Facebook API后出现一些错误而导致的失败。Facebook API不是事务性的——你不可能去“回滚”一个Facebook API调用。那个API调用的结果将永远不会回滚。

在那种情况下，当创建一个新的DbContextScope的时候，你可以传递DbContextScopeOption.ForceCreateNew的值作为joiningOption参数。这将创建一个不会加入环境上下文范围（如果存在一个的话）的DbContextScope：

[复制代码](javascript:void(0);)

public void RandomServiceMethod()

{

using (var dbContextScope = \_dbContextScopeFactory.Create(DbContextScopeOption.ForceCreateNew))

{

// 我们将创建一个新的范围（scope）,即使这个服务方法

// 被另一个已经创建了它自己的DbContextScope的服务方

// 法调用。我们将不会加入它。

// 我们的范围将创建新的DbContext实例并且不会重用

// 父范围的DbContext实例。

//[...]

// 由于我们强制创建了一个新的范围，这个对SaveChanges()

// 的调用将会持久化我们的修改——不管父范围（如果有的话）

// 是否成功保存了它的变化。

dbContextScope.SaveChanges();

}

}

[复制代码](javascript:void(0);)

这样处理的最大问题是服务方法将使用独立的DbContext实例而非业务事务中的其它DbContext。为了避免诡异的bug和维护恶魔，下面列出了一些要服从的基本规则：

**服务方法返回的持久化实体必须总是依附（Attach）在环境上下文DbContext上**

如果你强制创建一个新的DbContextScope而非加入一个已经存在的上下文环境DbContextScope,你的服务方法必须不能返回它在新的范围（scope)创建或者获取的实体。否则将导致巨大的复杂性。

调用你服务方法的客户端代码可能是一个创建了它自己的DbContextScope的服务方法。因此它期望它调用的所有服务方法都使用相同的环境上下文DbContextScope(这是使用环境上下文Context的立足点)。很自然的期望通过你的服务方法返回的实体都依附（attach)在环境上下文DbContext上。

相反，你也可以采取下面两种策略：

     不要返回持久化实体。这是最容易，最干净的方法。比如，如果你的服务方法创建一个新的领域对象，不要返回它，而是返回它的Id并且让客户端在它自己的DbContext上面加载这个实体（如果客户端真的需要这个实体的话）。

     如果你无论如何也要返回一个持久化实体的话，切换回环境上下文DbContext,加载实体并将其返回。

**在退出时，一个服务方法必须确保对持久化对象的所有修改都已经在父范围中重现**

如果你的服务方法强制创建了一个新的DbContextScope并且在这个新的范围里面修改了持久化对象，必须确保在返回的时候父范围（如果存在的话）能“看到”这些修改。

也就是说，如果父范围的DbContext实例已经加载了你修改过的实体在它的一级缓存中（ObjectStateManager),你的服务方法必须刷新这些实体以确保父范围不会使用这些对象的过时版本。

DbContextScope提供了一个快捷方法来帮助处理这个问题：

[复制代码](javascript:void(0);)

public void RandomServiceMethod(Guid accountId)

{

// 强制创建一个新范围（也就是说，我们将使用我们自己

// 的DbContext 实例）

using (var dbContextScope = \_dbContextScopeFactory.Create(DbContextScopeOption.ForceCreateNew))

{

var account = \_accountRepository.Get(accountId);

account.Disabled = true;

// 由于我们强制创建了一个新的范围，这将持久化我

// 们的变化到数据库而不管父范围的处理成功与否。

dbContextScope.SaveChanges();

// 如果这个方法的调用者已经加载过account对象到

// 它们的DbContext实例中，它们的版本现在已经变

// 得过时了。它们将看不到这个account已经被禁用

// 并且可能因此执行一些错误的逻辑。

// 因此需要确保我们的调用者的版本要保持更新。

dbContextScope.RefreshEntitiesInParentScope(new[] { account });

}

}

[复制代码](javascript:void(0);)

**为什么命名为DbContextScope而不是UnitOfWork(工作单元)？**

我写的DbContextScope的第一个版本确实被命名为UnitOfWork，这可以说是这种类型的组件最常用的名称。

但是当我尝试在现实程序中使用那个UnitOfWork组件的时候，我一直很困惑——我应该如何使用它和它真的做了什么——尽管我是那个研究，设计和实现了它的人并且我还对它能做什么以及如何工作都了如指掌。然而，我仍然很困难并且不得不倒退一步去仔细回想这个“unit of work”怎样关联我要尝试解决的实际问题：管理我的DbContext实例。

如果即使我——那个花了很多时间去研究，设计和实现了这个组件的人在尝试使用的时候都变得很困惑的话，要让其他人来容易的使用它——这恐怕是没什么希望了。

因此我将其重命名为DbContextScope并且突然所有的事情都变得清晰明朗了。

使用UnitOfWork最主要的问题我相信是在应用程序级别的，它通常没有什么意义。在一个更低的层次，比如数据库级别，一个“unit of work”是一个非常清晰并且具体的概念。下面是Martin Fowler对unit of work的定义：

*维护受业务影响的对象列表，并协调变化和并发问题的解决。*

在数据库级别，unit of work要表达的东西没有二义性。

然而在一个应用程序级别，一个”unit of work”是一个非常模糊的概念——它可能指所有东西，但又可能什么都不是。并且这个“unit of work”如何关联到EF是不清晰的——对于管理DbContext实例的问题，对于我们操作的持久化对象依附到正确的DbContext实例上的问题。

因此，任何开发人员在尝试使用一个”UnitOfWork”的时候都会搜索它的源代码去查看它究竟做了什么。工作单元（unit of work)模式的定义太过于模糊以至于在应用程序级别没什么用处。

实际上，对大部分应用程序，一个应用程序级别的“unit of work”甚至没有任何意义。许多应用程序在业务事务中不得不使用几个非事务性的服务，比如远程API或者非事务性的遗留组件。这些地方做出的修改不能被回滚。假装这些不存在是反效率的，迷惑的并且甚至更难写出正确的代码。

相反，DbContextScope刚好完成了它需要完成的工作，不多，不少。它没有假装成别的东西。并且我发现这个简单的更名有效的减少了使用这个组件的认知负荷和去验证是否正确的使用了它。

当然，将这个组件命名为DbContextScope就再也不能掩盖你的服务方法正在使用EF的事实。UnitOfWork是一个非常模糊的概念——它允许抽象在底层使用的持久化机制。从你的服务层中抽象EF是否是一件好事是一个另外争论——我们在这儿就不深入它了。

**直接去看看吧**

放在[GitHub上的源代码](https://github.com/mehdime/DbContextScope)包括了一个demo程序来演示大部分的使用场景

**DbContextScope是如何工作的**

[源代码](https://github.com/mehdime/DbContextScope)已经做了很好的注释并且我鼓励你通读它。另外，[Stephen Toub写的这篇放到ExecutionContext的优秀文章](http://blogs.msdn.com/b/pfxteam/archive/2012/06/15/executioncontext-vs-synchronizationcontext.aspx)是必读的——如果你想要完全理解DbContextScope中的环境上下文context模式是如何实现的话。

**延伸阅读**

EF团队的项目经理[Rowan Miller，他的个人博客](http://romiller.com/)，对于用EF开发项目的任何开发人员来说都是必须要去读的。

**额外资料**

**哪些地方不能创建你的DbContext实例**

在现实程序中经常看到的一个使用EF的反模式是将创建和释放DbContext的代码都放到数据访问方法里面（也就是在传统三层架构中的仓储方法里面）。它通常看起来像这样：

[复制代码](javascript:void(0);)

public class UserService : IUserService

{

private readonly IUserRepository \_userRepository;

public UserService(IUserRepository userRepository)

{

if (userRepository == null) throw new ArgumentNullException("userRepository");

\_userRepository = userRepository;

}

public void MarkUserAsPremium(Guid userId)

{

var user = \_userRepository.Get(userId);

user.IsPremiumUser = true;

\_userRepository.Save(user);

}

}

public class UserRepository : IUserRepository

{

public User Get(Guid userId)

{

using (var context = new MyDbContext())

{

return context.Set<User>().Find(userId);

}

}

public void Save(User user)

{

using (var context = new MyDbContext())

{

// [...]

// (要么将提供的实体依附在context上，要么从context加载它,

// 并且从提供的实体更新它的字段）

context.SaveChanges();

}

}

}

[复制代码](javascript:void(0);)

通过这样处理，你基本上失去了EF通过DbContext提供的每一个功能，包括它的一级缓存，它的标识映射（Identity map),它的工作单元（unit-of-work)，它的变更追踪和延迟加载功能。因为在上面的场景中，对于每一个数据库查询都将创建一个新的DbContext实例并且随后立即就被释放掉，因此阻碍了DbContext实例去跟踪你的整个业务事务范围内的数据的状态。

你有效的将EF简化为一个简单ORM框架：一个将你的对象与它在数据库中的关系表现映射的工具。

这种架构对于一些应用程序是说得通的。如果你工作在这样一个应用程序，无论如何你应当首先问你自己为什么要用EF。如果你要将它作为一个简单ORM框架并且不用它提供的任何主要功能，你可能使用一个轻量级的ORM框架（比如[Dapper](https://github.com/StackExchange/dapper-dot-net))会更好。因为它将会简化你的代码并且由于没有EF附加功能的开销而提供更好的性能。