عنوان: **تقسیم جدول در Entity Framework Code First**
نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۲/۰۳/۲۹ ۱:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework

سناریو هایی هستند که در آن ها، تعداد ستون‌های یک جدول، بیش از اندازه زیاد می‌شوند و یا آن جدول حاوی فیلدهایی هست که منابع زیادی مصرف می‌کنند، به مانند فیلدهای متنی طولانی یا عکس. معمولا متوجه می‌شویم که در اکثر مواقع، به هنگام واکشی اطلاعات آن جدول، احتیاجی به داده‌های آن فیلدها نداریم و با واکشی بی مورد آن ها، سربار اضافه ای به سیستم تحمیل می‌کنیم، چرا که این داده‌ها ، منابع حافظه ای ما را به هدر می‌دهند.
برای مثال، جدول Post مدل بلاگ را در نظر بگیرید که در آن دو فیلد Body و Image تعریف شده اند.فیلد Body از نوع nvarchar max و فیلد Image از نوع varbinary max است و بدیهی است که این دو داده، به هنگام واکشی حافظه‌ی زیادی مصرف می‌کنند.موارد بسیاری وجود دارند که ما به اطلاعات این دو فیلد احتیاجی نداریم از جمله: نمایش پست‌های پر بازدید، پسته هایی که اخیرا ارسال شده اند و اصولا ما فقط به چند فیلد جدول Post احتیاج داریم و نه همه‌ی آن ها.

```
namespace SplittingTableSample.DomainClasses
{
    public class Post
    {
        public virtual int Id { get; set; }
        public virtual string Title { get; set; }
        public virtual DateTime CreatedDate { get; set; }
        public virtual string  Body { get; set; }
        public virtual byte[] Image { get; set; }
    }
}
```

دلیل اینکه در مدل فوق، تمامی خواص به صورت virtual تعریف شده اند، فعال سازی پروکسی‌های ردیابی تغییر است. اگر دستور زیر را برای واکشی اطلاعات post با id=1 انجام دهیم:

```
using (var context = new MyDbContext())
            {
                var post = context.Posts.Find(1);
            }
```

خروجی زیر را در SQL Server Profiler مشاهده خواهید کرد:

```
exec sp_executesql N'SELECT TOP (2)
[Extent1].[Id] AS [Id],
[Extent1].[Title] AS [Title],
[Extent1].[CreatedDate] AS [CreatedDate],
[Extent1].[Body] AS [Body],
[Extent1].[Image] AS [Image]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[Id] = @p0',N'@p0 int',@p0=1
```

همان طور که مشاهده می‌کنید، با اجرای دستور فوق تمامی فیلدهای جدول Posts که id آن‌ها برابر 1 بود واکشی شدند، ولی من تنها به فیلدهای Id و Title آن احتیاج داشتم. خب شاید بگویید که من به سادگی با projection، این مشکل را حل می‌کنم و تنها از فیلد هایی که به آن‌ها احتیاج دارم، کوئری می‌گیرم. همه‌ی این‌ها درست، اما projection هم مشکلات خود را دارد،به صورت پیش فرض، نوع بدون نام بر می‌گرداند و اگر بخواهیم این گونه نباشد، باید مقادیر آن را به یک کلاس(مثلا viewmodel) نگاشت کنیم و کلی مشکل دیگر.
راه حل دیگری که برای حل این مشکل ارائه می‌شود و برای نرمال سازی جداول نیز کاربرد دارد این است که، جدول Posts را به دو جدول مجزا که با یکدیگر رابطه‌ی یک به یک دارند تقسیم کنیم، فیلدهای پر مصرف را در یک جدول و فیلدهای حجیم و کم

مصرف را در جدول دیگری تعریف کنیم و سپس یک رابطه‌ی یک به یک بین آن دو برقرار می‌کنیم.
به طور مثال این کار را بر روی جدول Posts ، به شکل زیر انجام شده است:

```
namespace SplittingTableSample.DomainClasses
{
    public class Post
    {
        public virtual int Id { get; set; }
        public virtual string Title { get; set; }
        public virtual DateTime CreatedDate { get; set; }
        public virtual PostMetaData PostMetaData { get; set; }
    }
}
namespace SplittingTableSample.DomainClasses
{
    public class PostMetaData
    {
        public virtual int PostId { get; set; }
        public virtual string Body { get; set; }
        public virtual byte[] Image { get; set; }
        public virtual Post Post { get; set; }
    }
}
```

همان طور که می‌بینید، خواص حجیم به جدول دیگری به نام PostMetaData منتقل شده و با تعریف خواص راهبری ارجاعی در هر دو کلاس،رابطه‌ی یک به یک بین آن‌ها برقرار شده است.جز الزامات تعریف روابط یک به یک این است که، با استفاده از Fluent API یا Data Annotations ، طرف‌های Depenedent و Principal، صریحا به EF معرفی شوند.

```
namespace SplittingTableSample.DomainClasses
{
    public class PostMetaDataConfig : EntityTypeConfiguration<PostMetaData>
    {
        public PostMetaDataConfig()
        {
            HasKey(x => x.PostId);
            HasRequired(x => x.Post).WithRequiredDependent(x => x.PostMetaData);
        }
    }
}
```

اولین نکته ای که باید به آن توجه شود، این است که در کلاس PostMetaData، قوانین پیش فرض EF برای تعیین کلید اصلی نقض شده است و به همین دلیل، صراحتا با استفاده از متد HasKey ، کلید اصلی به EF معرفی شده است. نکته‌ی مهم دیگری که به آن باید توجه شود این است که هر دو سر رابطه به صورت Required تعریف شده است. دلیل این موضوع هم با توجه به مطلبی که قرار است گفته شود،کمی جلوتر خواهید فهمید. حال اگر تعاریف DbSetها را نیز اصلاح کنیم و دستور زیر را اجرا کنیم:

```
var post = context.Posts.Find(1);
```

خروجی sql زیر را مشاهده خواهید کرد:

```
exec sp_executesql N'SELECT TOP (2)
[Extent1].[Id] AS [Id],
[Extent1].[Title] AS [Title],
[Extent1].[CreatedDate] AS [CreatedDate]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[Id] = @p0',N'@p0 int',@p0=1
```

خیلی خوب! دیگر خبری از فیلدهای اضافی Body و Image نیست. دلیل اینکه در اینجا join بین دو جدول مشاهده نمی‌شود،

قابلیت lazy loading است، که با virtual تعریف کردن خواص راهبری حاصل شده است. پس lazy loading در اینجا واقعا مفید است.

اما راه حل ذکر شده نیز کاملا بدون ایراد نیست. مشکل اساسی آن تعدد تعداد جداول آن است. آیا جدول Post ، واقعا احتیاج به چنین سطح نرمال سازی و تبدیل آن به دو جدول مجزا را داشت؟ مطمئنا خیر. آیا واقعا راه حلی وجود دارد که ما در سمت کدهای خود با دو موجودیت مجزا کار کنیم، در صورتی که در دیتابیس این دو موجودیت، ساختار یک جدول را تشکیل دهند. در اینجا روشی مطرح می‌شود به نام تقسیم جدول (Table Splitting).

برای انجام این کار فقط چند تنظیم ساده لازم است:

1) فیلدهای موجودیت مورد نظر را به موجودیت‌های کوچکتر، نگاشت می‌کنیم.

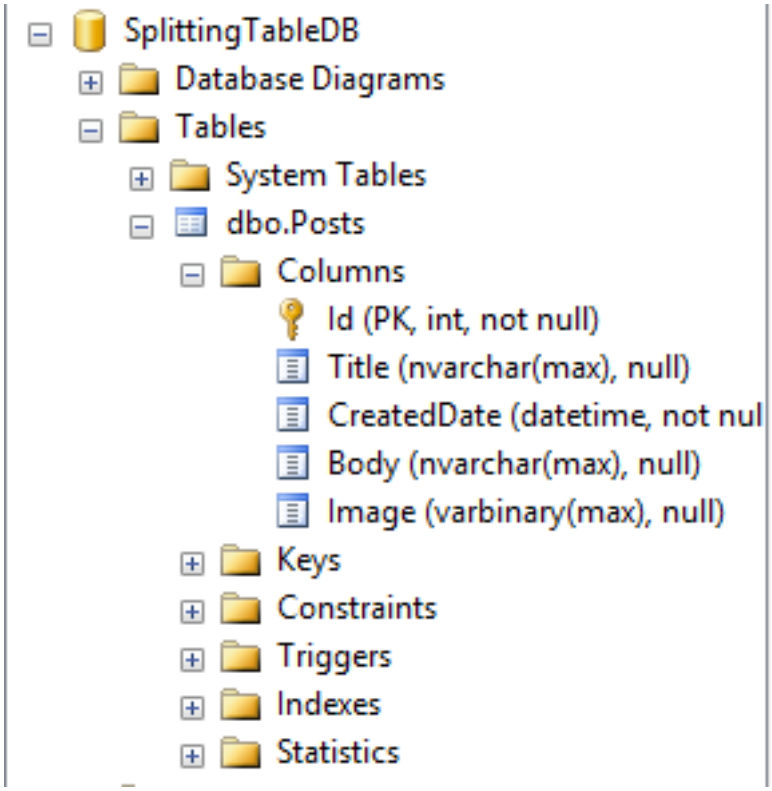
2) بین موجودیت‌های کوچک تر، رابطه‌ی یک به یک که هر دو سر رابطه Required هستند، رابطه برقرار می‌کنم.

3) با استفاده از Fluent API یا DataAnnotations، تمامی موجودیت‌ها را به یک نام در دیتابیس نگاشت می‌کنیم.

برای مثال، تنظیمات Fluent برای کلاس Post و PostMetaData که رابطه‌ی بین آن‌ها یک به یک است را مشاهده می‌کنید:

```
namespace SplittingTableSample.DomainClasses
{
    public class PostConfig : EntityTypeConfiguration<Post>
    {
        public PostConfig()
        {
            ToTable("Posts");
        }
    }
}
namespace SplittingTableSample.DomainClasses
{
    public class PostMetaDataConfig : EntityTypeConfiguration<PostMetaData>
    {
        public PostMetaDataConfig()
        {
            ToTable("Posts");
            HasKey(x => x.PostId);
            HasRequired(x => x.Post).WithRequiredDependent(x => x.PostMetaData);
        }
    }
}
```

نکته مهم این است که در هر دو کلاس(**حتی کلاس Post**) باید با استفاده از متد ToTable، کلاس‌ها را به **یک نام** در دیتابیس نگاشت کنیم. در نتیجه با استفاده از متد ToTable در هر دو موجودیت، آنها در دیتابیس به جدولی به نام Posts نگاشت خواهند شد. تصویر زیر پس از اجرای برنامه، بیان گر این موضوع خواهد بود.



اگر دستورات زیر را اجرا کنید:

```
var post = context.Posts.Find(1);
Console.WriteLine(post.PostMetaData.Body);
```

خروجی زیر را در SQL Server Profiler مشاهده خواهید کرد:
برای متد Find خروجی زیر:

```
exec sp_executesql N'SELECT TOP (2)
[Extent1].[Id] AS [Id],
[Extent1].[Title] AS [Title],
[Extent1].[CreatedDate] AS [CreatedDate]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[Id] = @p0',N'@p0 int',@p0=1
```

و برای post.PostMetaData.Body دستور sql زیر را مشاهده می‌کنید:

```
exec sp_executesql N'SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Body] AS [Body],
[Extent1].[Image] AS [Image]
FROM [dbo].[Posts] AS [Extent1]
WHERE [Extent1].[Id] = @EntityKeyValue1',N'@EntityKeyValue1 int',@EntityKeyValue1=1
```

دلیل این که در اینجا ،دو دستور sql به دیتابیس ارسال شده است، فعال بودن ویژگی lazy loading ،به دلیل virtual تعریف کردن خواص راهبری موجودیت‌ها است.
حال اگر بخواهیم با یک رفت و آمد به دیتابیس کلیه اطلاعات را واکشی کنیم، می‌توانیم از Eager Loading استفاده کنیم:

```
var post = context.Posts.Include(x => x.PostMetaData).SingleOrDefault(x => x.Id == 1);
```

که خروجی sql آن نیز به شکل زیر است:

```
SELECT
[Limit1].[Id] AS [Id],
[Limit1].[Title] AS [Title],
[Limit1].[CreatedDate] AS [CreatedDate],
[Extent2].[Id] AS [Id1],
[Extent2].[Body] AS [Body],
[Extent2].[Image] AS [Image]
FROM   (SELECT TOP (2) [Extent1].[Id] AS [Id], [Extent1].[Title] AS [Title], [Extent1].[CreatedDate] AS
[CreatedDate]
FROM [dbo].[Posts] AS [Extent1]
WHERE 1 = [Extent1].[Id] ) AS [Limit1]
LEFT OUTER JOIN [dbo].[Posts] AS [Extent2] ON [Limit1].[Id] = [Extent2].[Id]
```

در نتیجه با کمک این تکنیک توانستیم، با چند موجودیت، در قالب یک جدول رفتار کنیم و از مزیت‌های آن همچون lazy loading، نیز بهره مند شویم.

دریافت کدهای این بخش: SplittingTable-Sample.rar

نظرات خوانندگان

نویسنده: امیرحسین جلوداری
تاریخ: ۱۳۹۲/۰۳/۲۹ ۱۹:۱۳

ممنون ... برا من که خیلی مفید بود :)

نویسنده: محمد
تاریخ: ۱۳۹۳/۰۵/۰۱ ۲۲:۳۹

مطلبی که ارائه دادید در مورد ef6 صدق نمی‌کنه و خطای اینکه این تیبل نمی‌تواند دو کلید داشته باشد را می‌دهد و این در حالی هست که مدل رو با ef5 انجام میدهیم مشکلی نداره

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۵/۰۲ ۱۲:۳۶

PostId را در کلاس PostMetaData تبدیل کنید به Id.