عنوان: **روش‌هایی برای بهبود سرعت برنامه‌های مبتنی بر Entity framework**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۰۴ ۱۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework

در این مطلب تعدادی از شایع‌ترین مشکلات حین کار با Entity framework که نهایتا به تولید برنامه‌هایی کند منجر می‌شوند، بررسی خواهند شد.

مدل مورد بررسی

```
public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public virtual ICollection<BlogPost> BlogPosts { get; set; }
    }

    public class BlogPost
    {
        public int Id { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }

        [ForeignKey("UserId")]
        public virtual User User { get; set; }
        public int UserId { get; set; }
    }
```

کوئری‌هایی که در ادامه بررسی خواهند شد، بر روی رابطه‌ی one-to-many فوق تعریف شده‌اند؛ یک کاربر به همراه تعدادی مطلب منتشر شده.

مشکل 1: بارگذاری تعداد زیادی ردیف

```
var data = context.BlogPosts.ToList();
```

در بسیاری از اوقات، در برنامه‌های خود تنها نیاز به مشاهده‌ی قسمت خاصی از یک سری از اطلاعات، وجود دارند. به همین جهت بکارگیری متد ToList بدون محدود سازی تعداد ردیف‌های بازگشت داده شده، سبب بالا رفتن مصرف حافظه‌ی سرور و همچنین بالا رفتن میزان داده‌ای که هر بار باید بین سرور و کلاینت منتقل شوند، خواهد شد. یک چنین برنامه‌هایی بسیار مستعد به استثناهایی از نوع out of memory هستند.
راه حل: با استفاده از Skip و Take، مباحث صفحه‌ی بندی را اعمال کنید.

مشکل 2: بازگرداندن تعداد زیادی ستون

```
var data = context.BlogPosts.ToList();
```

فرض کنید View برنامه، در حال نمایش عناوین مطالب ارسالی است. کوئری فوق، علاوه بر عناوین، شامل تمام خواص تعریف شده‌ی دیگر نیز هست. یک چنین کوئری‌هایی نیز هربار سبب هدر رفتن منابع سرور می‌شوند.
راه حل: اگر تنها نیاز به خاصیت Content است، از Select و سپس ToList استفاده کنید؛ البته به همراه نکته 1.

```
var list = context.BlogPosts.Select(x => x.Content).Skip(15).Take(15).ToList();
```

مشکل 3: گزارشگیری‌هایی که بی‌شباهت به حمله‌ی به دیتابیس نیستند

```
foreach (var post in context.BlogPosts)
{
     Console.WriteLine(post.User.Name);
}
```

فرض کنید قرار است رکوردهای مطالب را نمایش دهید. در حین نمایش این مطالب، در قسمتی از آن باید نام نویسنده نیز درج شود. با توجه به رابطه‌ی تعریف شده، نوشتن post.User.Name به ازای هر مطلب، بسیار ساده به نظر می‌رسد و بدون مشکل هم کار می‌کند. اما ... اگر خروجی SQL این گزارش را مشاهده کنیم، به ازای هر ردیف نمایش داده شده، یکبار رفت و برگشت به بانک اطلاعاتی، جهت دریافت نام نویسنده یک مطلب وجود دارد.
این مورد به lazy loading مشهور است و در مواردی که قرار است با یک مطلب و یک نویسنده کار شود، شاید اهمیتی نداشته باشد. اما در حین نمایش لیستی از اطلاعات، بی‌شباهت به یک حمله‌ی شدید به بانک اطلاعاتی نیست.
راه حل: در گزارشگیری‌ها اگر نیاز به نمایش اطلاعات روابط یک موجودیت وجود دارد، از متد Include استفاده کنید تا Lazy loading لغو شود.

```
foreach (var post in context.BlogPosts.Include(x=>x.User))
```

مشکل 4: فعال بودن بی‌جهت مباحث ردیابی اطلاعات

```
var data = context.BlogPosts.ToList();
```

در اینجا ما فقط قصد داریم که لیستی از اطلاعات را دریافت و سپس نمایش دهیم. در این بین، هدف، ویرایش یا حذف اطلاعات این لیست نیست. یک چنین کوئری‌هایی مساوی هستند با تشکیل dynamic proxies مخصوص EF جهت ردیابی تغییرات اطلاعات (مباحث AOP توکار). EF توسط این dynamic proxies، محصور کننده‌هایی را برای تک تک آیتم‌های بازگشت داده شده از لیست تهیه می‌کند. در این حالت اگر خاصیتی را تغییر دهید، ابتدا وارد این محصور کننده (غشاء نامرئی) می‌شود، در سیستم ردیابی EF ذخیره شده و سپس به شیء اصلی اعمال می‌گردد. به عبارتی شیء در حال استفاده، هر چند به ظاهر post.User است اما در واقعیت یک User دارای روکشی نامرئی از جنس dynamic proxyهای EF است. تهیه این روکش‌ها، هزینه‌بر هستند؛ چه از لحاظ میزان مصرف حافظه و چه از نظر سرعت کار.
راه حل: در گزاشگیری‌ها، dynamic proxies را توسط متد AsNoTracking غیرفعال کنید:

```
var data = context.BlogPosts.AsNoTracking().Skip(15).Take(15).ToList();
```

مشکل 5: باز کردن تعداد اتصالات زیاد به بانک اطلاعاتی در طول یک درخواست

هر Context دارای اتصال منحصربفرد خود به بانک اطلاعاتی است. اگر در طول یک درخواست، بیش از یک Context مورد استفاده قرار گیرد، بدیهی است به همین تعداد اتصال باز شده به بانک اطلاعاتی، خواهیم داشت. نتیجه‌ی آن فشار بیشتر بر بانک اطلاعاتی و همچنین کاهش سرعت برنامه است؛ از این لحاظ که اتصالات TCP برقرار شده، هزینه‌ی بالایی را به همراه دارند.
روش تشخیص:

```
private void problem5MoreThan1ConnectionPerRequest()
        {
            using (var context = new MyContext())
            {
                var count = context.BlogPosts.ToList();
            }
        }
```

داشتن متدهایی که در آن‌ها کار وهله سازی و dispose زمینه‌ی EF انجام می‌شود (متدهایی که در آن‌ها new Context وجود دارد).

راه حل: برای حل این مساله باید از روش‌های تزریق وابستگی‌ها استفاده کرد. یک Context وهله سازی شده‌ی در طول عمر یک درخواست، باید بین وهله‌های مختلف اشیایی که نیاز به Context دارند، زنده نگه داشته شده و به اشتراک گذاشته شود.

مشکل 6: فرق است بین IList و IEnumerable

```
DataContext = from user in context.Users
                      where user.Id>10
                      select user;
```

خروجی کوئری LINQ نوشته شده از نوع IEnumerable است. در EF، هربار مراجعه‌ی مجدد به یک کوئری که خروجی IEnumerable دارد، مساوی است با ارزیابی مجدد آن کوئری. به عبارتی، یکبار دیگر این کوئری بر روی بانک اطلاعاتی اجرا خواهد شد و رفت و برگشت مجددی صورت می‌گیرد.
زمانیکه در حال تهیه‌ی گزارشی هستید، ابزارهای گزارشگیر ممکن است چندین بار از نتیجه‌ی کوئری شما در حین تهیه‌ی گزارش استفاده کنند. بنابراین برخلاف تصور، data binding انجام شده، تنها یکبار سبب اجرای این کوئری نمی‌شود؛ بسته به ساز و کار درونی گزارشگیر، چندین بار ممکن است این کوئری فراخوانی شود.
راه حل: یک ToList را به انتهای این کوئری اضافه کنید. به این ترتیب از نتیجه‌ی کوئری، بجای اصل کوئری استفاده خواهد شد و در این حالت تنها یکبار رفت و برگشت به بانک اطلاعاتی را شاهد خواهید بود.

مشکل 7: فرق است بین IQueryable و IEnumerable

خروجی IEnumerable، یعنی این عبارت را محاسبه کن. خروجی IQueryable یعنی این عبارت را درنظر داشته باش. اگر نیاز است نتایج کوئری‌ها با هم ترکیب شوند، مثلا بر اساس رابط کاربری برنامه، کاربر بتواند شرط‌های مختلف را با هم ترکیب کند، باید از ترکیب IQueryableها استفاده کرد تا سبب رفت و برگشت اضافی به بانک اطلاعاتی نشویم.

مشکل 8: استفاده از کوئری‌های Like دار

```
var list = context.BlogPosts.Where(x => x.Content.Contains("test"))
```

این نوع کوئری‌ها که در نهایت به Like در SQL ترجمه می‌شوند، سبب full table scan خواهند شد که کارآیی بسیار پایینی دارند. در این نوع موارد توصیه شده‌است که از روش‌های full text search استفاده کنید.

مشکل 9: استفاده از Count بجای Any

اگر نیاز است بررسی کنید مجموعه‌ای دارای مقداری است یا خیر، از Count>0 استفاده نکنید. کارآیی Any و کوئری SQL ایی که تولید می‌کند، به مراتب بیشتر و بهینه‌تر است از Count>0.

مشکل 10: سرعت insert پایین است

ردیابی تغییرات را خاموش کرده و از متد جدید AddRange استفاده کنید. همچنین افزونه‌هایی برای Bulk insert نیز موجود هستند.

مشکل 11: شروع برنامه کند است

می‌توان تمام مباحث نگاشت‌های پویای کلاس‌های برنامه به جداول و روابط بانک اطلاعاتی را به صورت کامپایل شده در برنامه ذخیره کرد . این مورد سبب بالا رفتن سرعت شروع برنامه خصوصا در حالتیکه تعداد جداول بالا است می‌شود.

نظرات خوانندگان

نویسنده: محمد شیران
تاریخ: ۱۳۹۳/۰۴/۰۴ ۱۷:۳۰

مطلب فوق العاده آموزنده ای بود. لطفا در خصوص نحوه استفاده از ساز و کار full text search در EF هم اگه روشی هست توضیح بفرمایید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۰۴ ۱۷:۳۵

- در کنفرانس techEd 2014 در جلسه « Entity Framework: Building Applications with Entity Framework 6 » کار با Full text search در EF 6، جزو مثال‌های مطرح شده‌است.
- روش دوم هم در اینجا « Full text search in Microsoft's Entity Framework ».
- روش سوم با استفاده از IDbCommandInterceptor در اینجا « Microsoft's Full Text Search in Entity Framework 6 »

نویسنده: فواد عبداللهی
تاریخ: ۱۳۹۳/۰۴/۰۵ ۱۹:۳۳

ممنون از مطلب مفید تون
من با راه حل شماره 5 و استفاده همزمان از addrange یا (modify update) و ... (بجز select, add, delete) همزمان مشکل دارم! اگر ممکنه یه sample در این رابطه معرفی کنید.
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۰۵ ۲۰:۳۹

```
((DbSet<Category>)_categories).AddRange(...);

// or in the Sample07Context

public IEnumerable<TEntity> AddThisRange<TEntity>(IEnumerable<TEntity> entities) where TEntity : class
{
    return ((DbSet<TEntity>)this.Set<TEntity>()).AddRange(entities);
}
```

نویسنده: ناظم
تاریخ: ۱۳۹۳/۰۵/۲۰ ۱۷:۴۷

سلام؛ **خروجی IEnumerable، یعنی این عبارت را محاسبه کن**
وقتی خروجی query مثلا در نوع IEnumerable ذخیره میشه تا وقتی مورد استفاده قرار نگرفته رفت و برگشتی به بانک صورت نگرفته مثل IQueryable :

```
private IEnumerable<Entity1> ienumerableEntites;
        private IQueryable<Entity1> queryablelEntities;

            ienumerableEntites = context.Entity1.Where(x=>x.EntityID>50);
            queryablelEntities = context.Entity1.Where(x => x.EntityID > 50);
```

منظورتون از جمله بالا چیست؟

نویسنده: وحید نصیری

تاریخ: ۱۳۹۳/۰۵/۲۰ ۱۸:۲۹

« بررسی Deferred execution یا بارگذاری به تاخیر افتاده »

نویسنده: Elham
تاریخ: ۱۳۹۳/۰۸/۳۰ ۱۷:۱۹

به نظر شما کوئری‌های پایین رو چطور میشه بهینه نوشت؟

```
List<Stat> allQuestion = (from a in TempClass.Stats
         where a.Person.PersonID == TempClass.ActiveUser.PersonID &&
               a.Subject.SubjectID == tileNumber
         select a).AsParallel().ToList();

int allQuestionCount = allQuestion.Count;

int correctCount = (from a in allQuestion
                    where a.Person.PersonID == TempClass.ActiveUser.PersonID &&
                          a.Subject.SubjectID == tileNumber
                    select a.CorrectQuestionCount).Sum();

int totalTime = (from a in allQuestion
                 where a.Person.PersonID == TempClass.ActiveUser.PersonID &&
                       a.Subject.SubjectID == tileNumber
                 select a.TotalTime).Sum();

double score = (from a in allQuestion
                where a.Person.PersonID == TempClass.ActiveUser.PersonID &&
                      a.Subject.SubjectID == tileNumber
                select a.Score).Sum();
```

50 بار دستورات بالا اجرا میشه و یک مکث حدودا 20 ثانیه‌ای داره

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۳۰ ۱۷:۴۹

چندبار رفت و برگشت به بانک اطلاعاتی را می‌شود تبدیل کرد به یکبار رفت و برگشت: « اعمال توابع تجمعی بر روی چند ستون در Entity framework »

نویسنده: علیرضا م
تاریخ: ۱۳۹۳/۰۹/۰۴ ۱۵:۱۳

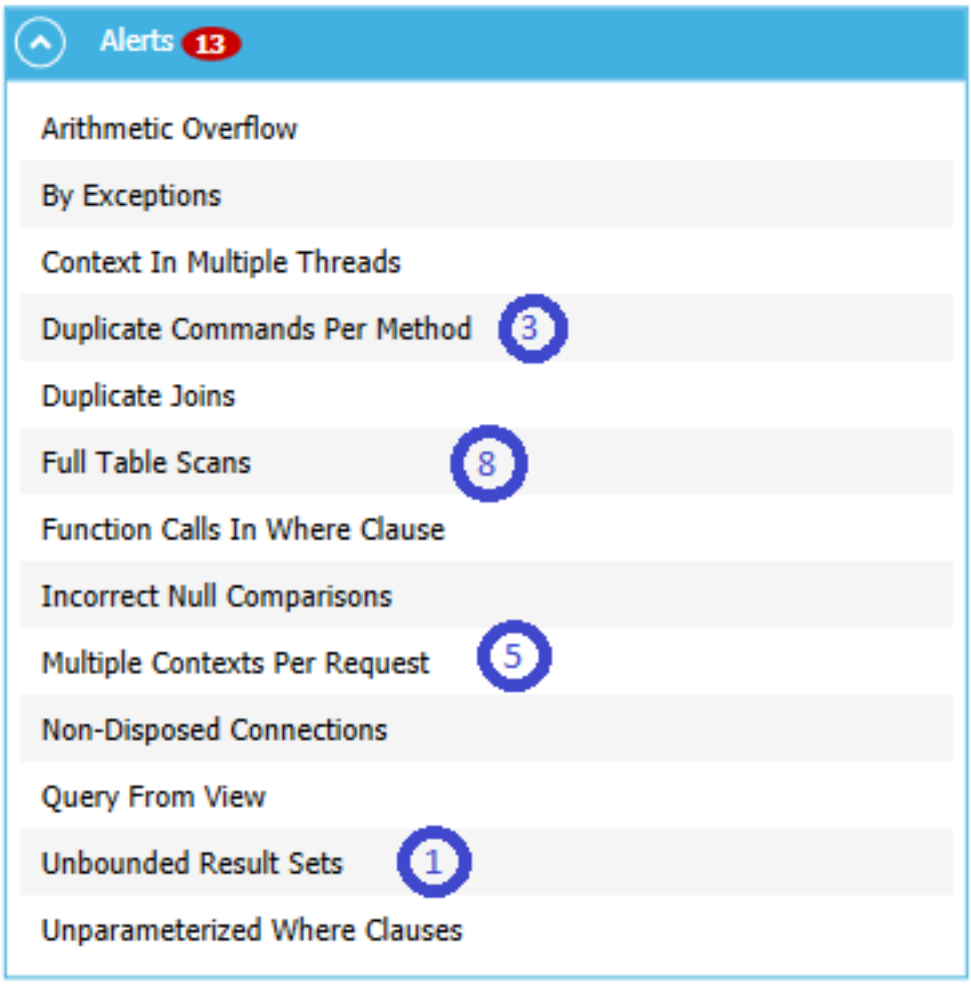
سلام
ایشان یکبار در ابتدا از ToList استفاده نموده اند. اعمال تجمعی که بعد از کویری اول نوشته شده است در حافظه محاسبه میشود یا در سمت بانک اطلاعاتی؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۹/۰۴ ۱۵:۲۸

- در حافظه.
- ولی در کل روش محاسبه‌ی sum این نیست که رکوردها را به همراه تمام ستون‌های جدول از بانک اطلاعاتی واکشی کرد و بعد در برنامه چند ستون انتخابی آن‌ها را جمع زد؛ زمانیکه خود بانک اطلاعاتی این توانایی را به نحو بهینه‌تری دارد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۱/۰۱ ۱۹:۴۰

اگر بخواهیم شماره‌ی نکات لیست شده‌ی در این مطلب را با برنامه‌ی DNTProfiler تطابق دهیم به شکل زیر خواهیم رسید:


Alerts 13
Arithmetic Overflow
By Exceptions
Context In Multiple Threads
Duplicate Commands Per Method 3
Duplicate Joins
Full Table Scans 8
Function Calls In Where Clause
Incorrect Null Comparisons
Multiple Contexts Per Request 5
Non-Disposed Connections
Query From View
Unbounded Result Sets 1
Unparameterized Where Clauses