عنوان: **بازنویسی سطح دوم کش برای Entity framework 6**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۶ ۱۶:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Performance, Caching

چندی قبل مطلبی را در مورد پیاده سازی سطح دوم کش در EF در این سایت مطالعه کردید . اساس آن مقاله‌ای بود که نحوه‌ی کش کردن اطلاعات حاصل از LINQ to Objects را بیان کرده بود (^). این مقاله پایه‌ی بسیاری از سیستم‌های کش مشابه نیز شده‌است (^ و ^ و ...).
مشکل مهم این روش عدم سازگاری کامل آن با EF است. برای مثال در آن تفاوتی بین Include(x=>x.Tags) و Include(x=>x.Users) وجود ندارد. به همین جهت در این نوع موارد، قادر به تولید کلید منحصربفردی جهت کش کردن اطلاعات یک کوئری مشخص نیست. در اینجا یک کوئری LINQ، به معادل رشته‌ای آن تبدیل می‌شود و سپس Hash آن محاسبه می‌گردد. این هش، کلید ذخیره سازی اطلاعات حاصل از کوئری، در سیستم کش خواهد بود. زمانیکه دو کوئری Include دار متفاوت EF، هش‌های یکسانی را تولید کنند، عملا این سیستم کش، کارآیی خودش را از دست می‌دهد. برای رفع این مشکل پروژه‌ی دیگری به نام EF cache ارائه شده‌است. این پروژه بسیار عالی طراحی شده و می‌تواند جهت ایده دادن به تیم EF نیز بکار رود. اما در آن فرض بر این است که شما می‌خواهید کل سیستم را در یک کش قرار دهید. وارد مکانیزم DBCommand و DataReader می‌شود و در آن‌جا کار کش کردن تمام کوئری‌ها را انجام می‌دهد؛ مگر آنکه به آن اعلام کنید از کوئری‌های خاصی صرفنظر کند.
با توجه به این مشکلات، روش بهتری برای تولید هش یک کوئری LINQ to Entities بر اساس کوئری واقعی SQL تولید شده توسط EF، پیش از ارسال آن به بانک اطلاعاتی به صورت زیر وجود دارد:

```
private static ObjectQuery TryGetObjectQuery<T>(IQueryable<T> source)
        {
            var dbQuery = source as DbQuery<T>;

            if (dbQuery != null)
            {
                const BindingFlags privateFieldFlags =
                    BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Public;

                var internalQuery =
                    source.GetType().GetProperty("InternalQuery", privateFieldFlags)
                        .GetValue(source);

                return
                    (ObjectQuery)internalQuery.GetType().GetProperty("ObjectQuery", privateFieldFlags)
                        .GetValue(internalQuery);
            }

            return null;
        }
```

این متد یک کوئری LINQ مخصوص EF را دریافت می‌کند و با کمک Reflection، اطلاعات درونی آن که شامل ObjectQuery اصلی است را استخراج می‌کند. سپس فراخوانی متد objectQuery.ToTraceString بر روی حاصل آن، سبب تولید SQL معادل کوئری LINQ اصلی می‌گردد. همچنین objectQuery امکان دسترسی به پارامترهای تنظیم شده‌ی کوئری را نیز میسر می‌کند. به این ترتیب می‌توان به معادل رشته‌ای منطقی‌تری از یک کوئری LINQ رسید که قابلیت تشخیص JOINها و متد Include نیز به صورت خودکار در آن لحاظ شده‌است.

این اطلاعات، پایه‌ی تهیه‌ی کتابخانه‌ی جدیدی به نام EFSecondLevelCache گردید. برای نصب آن کافی است دستور ذیل را در کنسول پاورشل نیوگت صادر کنید:

```
PM> Install-Package EFSecondLevelCache
```

سپس برای کش کردن کوئری معمولی مانند:

```
var products = context.Products.Include(x => x.Tags).FirstOrDefault();
```

می‌توان از متد جدید Cacheable آن به نحو ذیل استفاده کرد (این روش بسیار تمیزتر است از روش مقاله‌ی قبلی و امکان استفاده‌ی از انواع و اقسام متدهای EF را به صورت متداولی میسر می‌کند):

```
var products = context.Products.Include(x => x.Tags).Cacheable().FirstOrDefault(); // Async methods are supported too.
```

پس از آن نیاز است کدهای کلاس Context خود را نیز به نحو ذیل ویرایش کنید:

```
namespace EFSecondLevelCache.TestDataLayer.DataLayer
{
    public class SampleContext : DbContext
    {
        // public DbSet<Product> Products { get; set; }

        public SampleContext()
            : base("connectionString1")
        {
        }

        public override int SaveChanges()
        {
            return SaveAllChanges(invalidateCacheDependencies: true);
        }

        public int SaveAllChanges(bool invalidateCacheDependencies = true)
        {
            var changedEntityNames = getChangedEntityNames();
            var result = base.SaveChanges();
            if (invalidateCacheDependencies)
            {
               new EFCacheServiceProvider().InvalidateCacheDependencies(changedEntityNames);
            }
            return result;
        }

        private string[] getChangedEntityNames()
        {
            return this.ChangeTracker.Entries()
                .Where(x => x.State == EntityState.Added ||
                            x.State == EntityState.Modified ||
                            x.State == EntityState.Deleted)
                .Select(x => ObjectContext.GetObjectType(x.Entity.GetType()).FullName)
                .Distinct()
                .ToArray();
        }
    }
}
```

متد InvalidateCacheDependencies سبب می‌شود تا اگر تغییری در بانک اطلاعاتی رخداد، به صورت خودکار کش‌های کوئری‌های مرتبط غیر معتبر شوند و برنامه اطلاعات قدیمی را از کش نخواند.

کدهای کامل این پروژه را از مخزن کد ذیل می‌توانید دریافت کنید:

EFSecondLevelCache

پ.ن.
این کتابخانه هم اکنون در سایت جاری در حال استفاده است.

نظرات خوانندگان

نویسنده: اس حیدری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۹

برای داشتن دو یا چند Context و یا تغییر کانکشن Context می‌توان از این Cash استفاده کرد؟

چرا که کلید بر اساس معادل اسکیول عبارت Linq ایجاد می‌شود

نویسنده: ایمان دارابی
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۴۹

این هم کتابخانه خوبی هست. البته expire شدن کش را با استفاده از تگ هندل می‌کنه. خوبیش اینه بچ دیلیت و آپدید و امکانات دیگه هم داره.
می شه از تگ به صورت اتوماتیک با روش شما ایجاد کرد و از کش همین کتابخانه استفاده کرد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۵۵

رشته اتصالی هم در حین تولید کلید درنظر گرفته شده‌است . همچنین در صورت نیاز یک عبارت دلخواه را که به آن در اینجا saltKey گفته می‌شود، می‌توانید به رشته‌ی نهایی که از آن کلید تولید می‌شود، اضافه کنید. برای اینکار پارامتر EFCachePolicy را مقدار دهی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۵

در انتهای سطر دوم مطلب، به این کتابخانه اشاره شده‌است. این مشکلات را دارد:
- چون از روش LINQ to Objects برای تهیه معادل رشته‌ای کوئری درخواستی استفاده می‌کند (دقیقا این روش: ^) قادر نیست Includeها و جوین‌های EF را پردازش کند و در این حالت برای تمام جوین‌ها یک هش مساوی را در سیستم خواهید داشت.
- چون قادر نیست cache dependencies را از کوئری به صورت خودکار استخراج کند، شما نیاز خواهید داشت تا پارامتر تگ‌های آن‌را به صورت دستی به ازای هر کوئری تنظیم کنید. این‌کار به صورت خودکار در پروژه‌ی جاری انجام می‌شود. cache dependencies به این معنا است که کوئری جاری به چه موجودیت‌هایی در سیستم وابستگی دارد. از آن برای به روز رسانی کش استفاده می‌شود. برای مثال اگر یک کوئری به سه موجودیت وابستگی دارد، با تغییر یکی از آن‌ها، باید کش غیرمعتبر شده و در درخواست بعدی مجددا ساخته شود.

نویسنده: محمد عیدی مراد
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۲۲

ظاهرا در حالت Lazy Loading زمانی که آبجکتی از کش لود میشه، پراپرتی‌های Navigation استثنای زیر را صادر میکنن:
The ObjectContext instance has been disposed and can no longer be used for operations that require a connection

تیکه کدی که این ارور رو بر میگردونه:

```
var userInRoles = user.UserInRoles.Union(user.UsersSurrogate.Where(a => a.SurrogateFromDate != null &&
a.SurrogateToDate != null && a.SurrogateFromDate <= DateTime.Now && a.SurrogateToDate >=
DateTime.Now).SelectMany(a => a.UserInRoles));
  result = userInRoles.Any(a => a.Role.FormRoles.Any(b => b.IsActive && (b.Select && b.Form.SelectPath
!= null && b.Form.SelectPath.ToLower().Split(',').Contains(roleName))));
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۴۴

Lazy loading با کش سازگاری ندارد؛ چون اتصال اشیاء موجود در کش از context قطع شده‌اند. در بار اول فراخوانی یک کوئری که قرار است کش شود، از context و دیتابیس استفاده می‌شود. اما در بارهای بعد دیگر به context و دیتابیس مراجعه نخواهد شد. تمام اطلاعات از کش سیستم بارگذاری می‌شوند و حتی یک کوئری اضافی نیز به بانک اطلاعاتی ارسال نخواهد شد.
به همین جهت در این موارد باید از متد Include برای eager loading اشیایی که نیاز دارید استفاده کنید.

نویسنده: اس حیدری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۱:۲۹

همچنین اتوماتیک بودن Cash به ازای کلیه Queryها هم می‌تواند یک آپشن در نظر گرفته شود و در مواری که دسترسی به کوئری‌های داخلی نیست مفید واقع شود.

مثلا اگر برای اعتبار سنجی کاربر از Identity استفاده شود عملا نمی‌توان به کوئری‌های داخلی Identity دسترسی پیدا کرد و نیاز است که آن کوئری‌ها Cash شود، چرا که بسیار پرکاربرد می‌باشند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۱:۳۷

- کش سطح دوم نباید برای کش کردن اطلاعات خصوصی استفاده شود؛ یا کلا اطلاعاتی که نیاز به سطح دسترسی دارند. هدف آن کش کردن اطلاعات عمومی و پر مصرف است. اطلاعات خاص یک کاربر نباید کش شوند.
- در تمام سیستم‌ها، برای مواردی که به کوئری‌های آن دسترسی ندارید تا متد Cacheable را به آن‌ها اضافه کنید، نتیجه‌ی کوئری‌ها را باید خودتان از طریق روش‌های متداول کش کنید (مانند کلاس CacheManager مطلب یاد شده).

نویسنده: امین کاشانی
تاریخ: ۱۳۹۳/۱۲/۰۸ ۱:۲۵

من در کد زیر expiretime را 60 ثانیه گذاشتم .ولی در هربار فراخوانی در بازه زمانی 60 ثانیه از کش نمی‌خواند و دیتا از دیتابیس برمی گرداند. ایراد کار کجاست؟ ولی بدون نوشتن پارامتر زمان در متد cacheable کش درست عمل می‌کند.

```
public async Task<IList<Bestankaran>> GetBestankaran()
        {
         EFCachePolicy expirationTime = new EFCachePolicy { AbsoluteExpiration = new
DateTime().AddSeconds(60) };

            var result =
                Task.Run(() =>
                    _bestankaran.Cacheable(expirationTime).ToListAsync());

            return await result;

        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۲/۰۸ ۱:۳۲

- زمانیکه از متد ToListAsync استفاده می‌کنید، نیازی به استفاده از Task.Run نیست. اطلاعات بیشتر
- بجای new DateTime باید از DateTime.Now استفاده کنید.