عنوان: ردیابی تغییرات در Entity Framework، بخش اول
نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۲/۰۳/۱۸ ۰:۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework

همان طور که می‌دانید، Entity Framework تغییراتی را که بر روی اشیا انجام می‌دهید، ردیابی می‌کند. بدیهی است که EF از طریق ردیابی این تغییرات است که می‌تواند تغییرات انجام شده را شناسایی کند و آن‌ها را در مواقع مورد نیاز مانند ذخیره‌ی تغییرات (DbContext.SaveChanges)، بر روی پایگاه داده اعمال کند. شما می‌توانید به اطلاعات این ردیاب تغییر و اعمال مرتبط به آن از طریق ویژگی DbContext.ChangeTracker دسترسی پیدا کنید.
در این مقاله بیشتر سعی به بررسی مفاهیم ردیابی و روش هایی که EF برای ردیابی تغییرات استفاده می‌کند، بسنده می‌کنم و بررسی APIهای مختلف آن را به مقاله ای دیگر موکول می‌کنم.
به طور کلی EF از دو روش برای ردیابی تغییرات رخ داده شده در اشیا استفاده می‌کند:
1) ردیابی تغییر عکس فوری! (Snapshot change tracking)
2) پروکسی‌های ردیابی تغییر (Change tracking proxies)

ردیابی تغییر عکس فوری

به نظر من، اسم مناسبی برای این روش انتخاب کرده اند و دقیقا بیان گر کاری است که EF انجام می‌دهد. در حالت عادی کلاس‌های دامین ما یا همان کلاس‌های POCO، هیچ منطق و کدی را برای مطلع ساختن EF از تغییراتی که در آن‌ها رخ می‌دهد پیاده سازی نکرده اند. چون هیچ راهی برای EF، برای مطلع شدن از تغییرات رخ داده وجود ندارد، EF راه جالبی را بر می‌گزیند. EF هر گاه شیئی را می‌بیند از مقادیر ویژگی‌های آن یک عکس فوری می‌گیرد! و آن‌ها را در حافظه ذخیره می‌کند.این عمل هنگامی که یک شی از پرس و جو (query) حاصل می‌شود، و یا شیئی را به DbSet اضافه می‌کنیم رخ می‌دهد.
زمانی که EF می‌خواهد بفهمد که چه تغییراتی رخ داده است، مقادیر کنونی موجود در کلیه اشیا را اسکن می‌کند و با مقادیری که در عکس فوری ذخیره کرده است مقایسه می‌کند و متوجه تغییرات رخ داده می‌شود. این فرآیند اسکن کردن کلیه اشیا زمانی رخ می‌دهد که متد DetectChanges ویژگی DbSet.ChangeTracker صدا زده شود.

پروکسی‌های ردیابی تغییر

پروکسی‌های ردیابی تغییر، مکانیزم دیگری برای ردیابی تغییرات EF است و به EF این اجازه را می‌دهد تا از تغییرات رخ داده، مطلع شود.
اگر به یاد داشته باشید در مباحث Lazy loading نیز از واژه پروکسی‌های پویا استفاده شد. پروکسی‌های ردیابی تغییر نیز با استفاده از همان مکانیزم کار می‌کنند و علاوه بر فراهم کردن Lazy loading ،این امکان را می‌دهند تا تغییرات را به Context انتقال دهند.
برای استفاده از پروکسی‌های ردیابی تغییر، شما باید ساختار کلاس‌های خود را به گونه ای تغییر دهید، تا EF بتواند در زمان اجرا، نوع پویایی را که هریک، از کلاس‌های POCO شما مشتق می‌شوند ایجاد کند، و تک تک ویژگی‌های آن‌ها را تحریف (override) کند.
این نوع پویا که به عنوان پروکسی پویا نیز شناخته می‌شود، منطقی را در ویژگی‌های تحریف شده شامل می‌شود، تا EF را از تغییرات صورت گرفته در ویژگی هایش مطلع سازد.
برای بیان ادامه‌ی مطلب، من مدل یک دفترچه تلفن ساده را به شرح زیر در نظر گرفتم که روابط مهم و اساسی در آن در نظر گرفته شده است.

```
namespace EntitySample1.DomainClasses
{
    public class Person
    {
        public int Id { get; set; }
```

```
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public virtual PersonInfo PersonInfo { get; set; }
        public virtual ICollection<PhoneNumber> PhoneNumbers { get; set; }
        public virtual ICollection<Address> Addresses { get; set; }
    }
}
```

```
namespace EntitySample1.DomainClasses
{
    public class PersonInfo
    {
        public int Id { get; set; }
        public string Note { get; set; }
        public string Major { get; set; }
    }
}
```

```
namespace EntitySample1.DomainClasses
{
    public enum PhoneType
    {
        Home,
        Mobile,
        Work
    }

    public class PhoneNumber
    {
        public int Id { get; set; }
        public string Number { get; set; }
        public PhoneType PhoneType { get; set; }
        public virtual Person Person { get; set; }
    }
}
```

```
namespace EntitySample1.DomainClasses
{
    public class Address
    {
        public int Id { get; set; }
        public string City { get; set; }
        public string Street { get; set; }
        public virtual ICollection<Person> Persons { get; set; }
    }
}
```

طبق کلاس‌های فوق ، ما تعدادی شخص ، اطلاعات شخص ، شماره تلفن و آدرس داریم. رابطه‌ی بین شخص و اطلاعات آن شخص یک به یک، شخص و آدرس چند به چند و شخص با شماره تلفن یک به چند است. همچنین به این نکته توجه داشته باشید که کلیه کلاس‌های فوق به صورت public تعریف، و کلیه خواص راهبری (navigation properties) به صورت virtual تعریف شده اند. دلیل این کار هم این است که این دو مورد، جز الزامات، برای فعال سازی Lazy loading هستند.
تعریف کلاس context نیز به شکل زیر است:

```
namespace EntitySample1.DataLayer
{
    public class PhoneBookDbContext : DbContext
    {
        public DbSet<Person> Persons { get; set; }
        public DbSet<PhoneNumber> PhoneNumbers { get; set; }
        public DbSet<Address> Addresses { get; set; }

    }
}
```

استفاده از ردیابی تغییر عکس فوری

ردیابی تغییر عکس فوری، وابسته به این است که EF بفهمد، چه زمانی تغییرات رخ داده است. رفتار پیش فرض DbContext API ، این هست که به صورت خودکار بازرسی لازم را در نتیجه‌ی رخدادهای DbContext انجام دهد. DetectChanges تنها اطلاعات مدیریت حالت context، که وظیفه‌ی انعکاس تغییرات صورت گرفته به پایگاه داده را دارد، به روز نمی‌کند، بلکه اصلاح رابطه(ralationship) ترکیبی از خواص راهبری مرجع ، مجموعه ای و کلیدهای خارجی را انجام می‌دهد. این خیلی مهم خواهد بود که درک روشنی داشته باشیم از این که چگونه و چه زمانی تغییرات تشخیص داده می‌شوند،چه چیزی باید از آن انتظار داشته باشیم و چگونه کنترلش کنیم.

چه زمانی تشخیص خودکار تغییرات اجرا می‌شود؟

متد DetectChanges کلاس ObjectContext، از EF نسخه‌ی 4 به عنوان بخشی از الگوی ردیابی تغییر عکس فوری اشیای POCO ،در دسترس بوده است. تفاوتی که در مورد DataContext.ChangeTracker.DetectChanges(در حقیقت ObjectContext.DetectChanges فراخوانی می‌شود) وجود دارد این است که، رویدادهای خیلی بیشتری وجود دارند که به صورت خودکار DetectChanges را فراخوانی می‌کنند.

لیستی از متدهایی که باعث انجام عمل تشخیص تغییرات (DetectChanges)، می‌شوند را در ادامه مشاهده می‌کنید:

- DbSet.Add
- DbSet.Find
- DbSet.Remove
- DbSet.Local
- DbSet.SaveChanges
- فراخوانی Linq Query از DbSet
- DbSet.Attach
- DbContext.GetValidationErrors
- DbContext.Entry
- DbChangeTracker.Entries

کنترل زمان فراخوانی DetectChanges

بیشترین زمانی که EF احتیاج به فهمیدن تغییرات دارد، در زمان SaveChanges است، اما حالت‌های زیاد دیگری نیز هست. برای مثال، اگر ما از ردیاب تغییرات، درخواست وضعیت فعلی یک شی را بکنیم،EF احتیاج به اسکن کردن و بررسی تغییرات رخ داده را دارد. همچنین وضعیتی را در نظر بگیرید که شما از پایگاه داده یک شماره تلفن را واکشی می‌کنید و سپس آن را به مجموعه شماره تلفن‌های یک شخص جدید اضافه می‌کنید.آن شماره تلفن اکنون تغییر کرده است، چرا که انتساب آن به یک شخص جدید،خاصیت PersonId آن را تغییر داده است. ولی EF برای اینکه بفهمد تغییر رخ داده است(یا حتی نداده است) ، احتیاج به اسکن کردن همه‌ی اشیا Person دارد.

بیشتر عملیاتی که بر روی DbContext API انجام می‌دهید، موجب فراخوانی DetectChanges می‌شود. در بیشتر موارد DetectChanges به اندازه کافی سریع هست تا باعث ایجاد مشکل کارایی نشود. با این حال ممکن است ، شما تعداد خیلی زیادی اشیا در حافظه داشته باشید، و یا تعداد زیادی عملیات در DbContext ، در مدت خیلی کوتاهی انجام دهید، رفتار تشخیص خودکار تغییرات ممکن است، باعث نگرانی‌های کارایی شود. خوشبختانه گزینه ای برای خاموش کردن رفتار تشخیص خودکار تغییرات وجود دارد و هر زمانی که می‌دانید لازم است، می‌توانید آن را به صورت دستی فراخوانی کنید.

EF بر مبنای این فرض ساخته شده است که شما ، در صورتی که در فراخوانی آخرین API، موجودیتی تغییر پیدا کرده است، قبل از فراخوانی API جدید، باید DetectChanges صدا زده شود. این شامل فراخوانی DetectChanges، قبل از اجرای هر query نیز می‌شود.اگر این عمل ناموفق یا نابجا انجام شود،ممکن است عواقب غیر منتظره ای در بر داشته باشد. DbContext انجام این وظیفه را بر عهده گرفته است و به همین دلیل به طور پیش فرض تشخیص تغییرات خودکار آن فعال است.

نکته: تشخیص اینکه چه زمانی احتیاج به فراخوانی DetectChanges است،آن طور که ساده و بدیهی به نظر می‌آید نیست. تیم EF شدیدا توصیه کرده اند که فقط، وقتی با مشکلات عدم کارایی روبرو شدید، تشخیص تغییرات را به حالت دستی در بیاورید.همچنین توصیه شده که در چنین مواقعی، تشخیص خودکار تغییرات را فقط برای قسمتی از کد که با کارایی پایین مواجه شدید خاموش کنید و پس از اینکه اجرای آن قسمت از کد تمام شد،دوباره آن را روشن کنید.

برای خاموش یا روشن کردن تشخیص خودکار تغییرات، باید متغیر بولین
DbContext.Configuration.AutoDetectChangesEnabled را تنظیم کنید.
در مثال زیر، ما در متد ManualDetectChanges، تشخیص خودکار تغییرات را خاموش کرده ایم و تاثیرات آن را بررسی کرده ایم.

```
private static void ManualDetectChanges()
        {
            using (var context = new PhoneBookDbContext())
            {
                context.Configuration.AutoDetectChangesEnabled = false; // turn off Auto Detect Changes

                var p1 = context.Persons.Single(p => p.FirstName == "joe");

                p1.LastName = "Brown";

                Console.WriteLine("Before DetectChanges: {0}", context.Entry(p1).State);

                context.ChangeTracker.DetectChanges(); // call detect changes manually

                Console.WriteLine("After DetectChanges: {0}", context.Entry(p1).State);
            }
        }
```

در کدهای بالا ابتدا تشخیص خودکار تغییرات را خاموش کرده ایم و سپس یک شخص با نام joe را از دیتابیس فراخواندیم و سپس نام خانوادگی آن را به Brown تغییر دادیم. سپس در خط بعد، وضعیت فعلی موجودیت p1 را از context جاری پرسیدیم. در خط بعدی، DetectChanges را به صورت دستی صدا زده ایم و دوباره همان پروسه را برای به دست آوردن وضیعت شی p1، انجام داده ایم. همان طور که می‌بینید ، برای به دست آوردن وضعیت فعلی شی مورد نظر از متد Entry متعلق به ChangeTracker API استفاده می‌کنیم، که در آینده مفصل در مورد آن بحث خواهد شد. اگر شما متد Main را با صدا زدن ManualDetectChanges ویرایش کنید ، خروجی زیر را مشاهده خواهید کرد:

```
Before DetectChanges: Unchanged
After DetectChanges: Modified
```

همان طور که انتظار می‌رفت، به دلیل خاموش کردن تشخیص خودکار تغییرات، context قادر به تشخیص تغییرات صورت گرفته در شی p1 نیست، تا زمانی که متد DetectChanges را به صورت دستی صدا بزنیم. دلیل این که در دفعه اول، ما نتیجه‌ی غلطی مشاهده می‌کنیم، این است که ما قانون را نقض کرده ایم و قبل از صدا زدن هر API ، متد DetectChanges را صدا نزده ایم.
خوشبختانه چون ما در اینجا وضعیت یک شی را بررسی کردیم، با عوارض جانبی آن روبرو نشدیم.

نکته : به این نکته توجه داشته باشید که متد Entry به صورت خودکار، DetectChanges را فراخوانی می‌کند. برای اینکه دانسته بخواهیم این رفتار را غیر فعال کنیم، باید AutoDetectChangesEnabled را غیر فعال کنیم.
در مثال فوق ،خاموش کردن تشخیص خودکار تغییرات، برای ما مزیتی به همراه نداشت و حتی ممکن بود برای ما دردسر ساز شود. ولی حالتی را در نظر بگیرید که ما یک سری API را فراخوانی می‌کنیم ،بدون این که در این بین ،در حالت اشیا تغییری ایجاد کنیم.در نتیجه می‌توانیم از فراخوانی‌های بی جهت DetectChanges جلوگیری کنیم.

در متد AddMultiplePersons مثال بعدی، این کار را نشان داده ام:

```
private static void AddMultiplePerson()
        {
            using (var context = new PhoneBookDbContext())
            {
                context.Configuration.AutoDetectChangesEnabled = false;

                context.Persons.Add(new Person
                    {
                        FirstName = "brad",
                        LastName = "watson",
                        BirthDate = new DateTime(1990, 6, 8)
                    });

                context.Persons.Add(new Person
                {
                    FirstName = "david",
                    LastName = "brown",
                    BirthDate = new DateTime(1990, 6, 8)
                });

                context.Persons.Add(new Person
                {
                    FirstName = "will",
                    LastName = "smith",
                    BirthDate = new DateTime(1990, 6, 8)
                });

                context.SaveChanges();

            }
        }
```

در مثال بالا ما از فراخوانی چهار DetectChanges غیر ضروری که شامل DbSet.Add و SaveChanges می‌شود، جلوگیری کرده ایم.

استفاده از DetectChanges برای فراخوانی اصلاح رابطه

DetectChanges همچنین مسئولیت انجام اصلاح رابطه ، برای هر رابطه ای که تشخیص دهد تغییر کرده است را دارد.اگر شما بعضی از روابط را تغییر دادید و مایل بودید تا همه‌ی خواص راهبری و خواص کلید خارجی را منطبق کنید، DetectChanges این کار را برای شما انجام می‌دهد. این قابلیت می‌تواند برای سناریوهای data-binding که در آن ممکن است در رابط کاربری(UI) یکی از خواص راهبری (یا حتی یک کلید خارجی) تغییر کند، و شما بخواهید که خواص دیگری این رابطه به روز شوند و تغییرات را نشان دهند، مفید واقع شود.
متد DetectRelationshipChanges در مثال زیر از DetectChanges برای انجام اصلاح رابطه استفاده می‌کند.

```
private static void DetectRelationshipChanges()
        {
            using (var context = new PhoneBookDbContext())
            {

                var phone1 = context.PhoneNumbers.Single(x => x.Number == "09351234567");

                var person1 = context.Persons.Single(x => x.FirstName == "will");

                person1.PhoneNumbers.Add(phone1);

                Console.WriteLine("Before DetectChanges: {0}", phone1.Person.FirstName);

                context.ChangeTracker.DetectChanges(); // ralationships fix-up

                Console.WriteLine("After DetectChanges: {0}", phone1.Person.FirstName);
            }
        }
```

در اینجا ابتدا ما شماره تلفنی را از دیتابیس لود می‌کنیم. سپس شخص دیگری را نیز با نام will از دیتابیس می‌خوانیم. قصد داریم

شماره تلفن خوانده شده را به این شخص نسبت دهیم و مجموعه شماره تلفن‌های وی اضافه کنیم و ما این کار را با افزودن phone1 به مجموعه شماره تلفن‌های person1 انجام داده ایم. چون ما از اشیای POCO استفاده کرده ایم،EF نمی‌فهمد که ما این تغییر را ایجاد کرده ایم و در نتیجه کلید خارجی PersonId شی phone1 را اصلاح نمی‌کند. ما می‌توانیم تا زمانی صبر کنیم تا متدی مثل SaveChanges، متد DetectChanges را فراخوانی کند،ولی اگر بخواهیم این عمل در همان لحظه انجام شود، می‌توانیم DetectChanges را دستی صدا بزنیم.

اگر ما متد Main را با اضافه کردن فرخوانی DetectRealtionShipsChanges تغییر بدهیم و آن را اجرا کنیم، نتیجه زیر را مشاهده می‌کنید:

```
Before DetectChanges: david
After DetectChanges: will
```

تا قبل از فراخوانی تشخیص تغییرات(DetectChanegs)، هنوز phone1 منتسب به شخص قدیمی(david) بوده، ولی پس از فراخوانی DetectChanges ، اصلاح رابطه رخ داده و همه چیز با یکدیگر منطبق می‌شوند.

فعال سازی و کار با پروکسی‌های ردیابی تغییر

اگر پروفایلر کارایی شما، فراخوانی‌های بیش از اندازه DetectChnages را به عنوان یک مشکل شناسایی کند، و یا شما ترجیح می‌دهید که اصلاح رابطه به صورت بلادرنگ صورت گیرد ، ردیابی تغییر پروکسی‌های پویا، به عنوان گزینه ای دیگر مطرح می‌شود.فقط با چند تغییر کوچک در کلاس‌های POCO، EF قادر به ساخت پروکسی‌های پویا خواهد بود.پروکسی‌های ردیابی تغییر به EF اجازه ردیابی تغییرات در همان لحظه ای که ما تغییری در اشیای خود می‌دهیم را می‌دهند و همچنین امکان انجام اصلاح رابطه را در هر زمانی که تغییرات روابط را تشخیص دهد، دارد.
برای اینکه پروکسی ردیابی تغییر بتواند ساخته شود، باید قوانین زیر رعایت شود:

- کلاس باید public باشد و seald نباشد.
- همه‌ی خواص(properties) باید virtual تعریف شوند.
- همه‌ی خواص باید getter و setter با سطح دسترسی public داشته باشند.
- همه‌ی خواص راهبری مجموعه ای باید نوعشان، از نوع ICollection<T> تعریف شوند.

کلاس Person مثال خود را به گونه ای بازنویسی کرده ایم که تمام قوانین فوق را پیاده سازی کرده باشد.

نکته: توجه داشته باشید که ما دیگر در داخل سازنده کلاس ،کدی نمی‌نویسیم و منطقی که باعث نمونه سازی اولیه خواص راهبری می‌شدند، را پیاده سازی نمی‌کنیم. این پروکسی ردیاب تغییر، همه‌ی خواص راهبری مجموعه ای را تحریف کرده و ار نوع مجموعه ای مخصوص خود(EntityCollection<T>) استفاده می‌کند. این نوع مجموعه ای، هر تغییری که در این مجموعه صورت می‌گیرد را زیر نظر گرفته و به ردیاب تغییر گزارش می‌دهد. اگر تلاش کنید تا نوع دیگری مانند List<T> که معمولا در سازنده کلاس از آن استفاده می‌کردیم را به آن انتساب دهیم، پروکسی، استثنایی را پرتاب می‌کند.

```
namespace EntitySample1.DomainClasses
{
    public class Person
    {
        public virtual int Id { get; set; }
        public virtual string FirstName { get; set; }
        public virtual string LastName { get; set; }
        public virtual DateTime BirthDate { get; set; }
        public virtual PersonInfo PersonInfo { get; set; }
        public virtual ICollection<PhoneNumber> PhoneNumbers { get; set; }
        public virtual ICollection<Address> Addresses { get; set; }
    }
}
```

همان طور که در مباحث مربوط به Lazy loading نیز مشاهده کردید،EF زمانی پروکسی‌های پویا را برای یک کلاس ایجاد می‌کند که یک یا چند خاصیت راهبری آن با virtual علامت گذاری شده باشند.آن پروکسی‌ها که از کلاس مورد نظر، مشتق شده اند، به خواص راهبری virtual امکان می‌دهند تا به صورت lazy لود شوند.پروکسی‌های ردیابی تغییر نیز به همان شکل در زمان اجرا ایجاد می‌شوند، با این تفاوت که این پروکسی ها، امکانات بیشتری دارند.

با این که احتیاجات رسیدن به پروکسی‌های ردیابی تغییر خیلی ساده هستند، اما ساده‌تر از آن ها، فراموش کردن یکی از آن هاست.حتی از این هم ساده‌تر می‌شود که در آینده تغییری در آن کلاس‌ها ایجاد کنید و ناخواسته یکی از آن قوانین را نقض کنید.به این خاطر، فکر خوبیست که یک آزمون واحد نیز اضافه کنیم تا مطمئن شویم که EF توانسته، پروکسی ردیابی تغییر را ایجاد کند یا نه.

در مثال زیر یک متد نوشته شده که این مورد را مورد آزمایش قرار می‌دهد. همچنین فراموش نکنید که فضای نام System.Data.Object.DataClasses را به usingهای خود اضافه کنید.

```
private static void TestForChangeTrackingProxy()
        {
            using (var context = new PhoneBookDbContext())
            {
                var person = context.Persons.First();
                var isProxy = person is IEntityWithChangeTracker;
                Console.WriteLine("person is a proxy: {0}", isProxy);
            }
        }
```

زمانی که EF ، پروکسی پویا برای ردیابی تغییر ایجاد می‌کند، اینترفیس IentityWithChangeTrackerرا پیاده سازی خواهد کرد.متد تست در مثال بالا، نمونه ای از Person را با دریافت آن از دیتابیس ایجاد می‌کند و سپس آن را با اینترفیس ذکر شده چک می‌کند تا مطمئن شود که Person ، توسط پروکسی ردیابی تغییر احاطه شده است. این نکته را نیز به یاد داشته باشید که چک کردن این که EF ، کلاس پروکسی ای که از کلاس ما مشتق شده است ایجاد کرده است یا نه،کفایت نمی‌کند، چرا که پروکسی‌های Lazy Loading نیز چنین کاری انجام می‌دهند. در حقیقت آن چیزی که سبب می‌شود EF به تغییرات صورت گرفته به صورت بلادرنگ گوش دهد،حضور IEntityWithChangeTracker است.

اکنون متد ManualDetectChanges را که کمی بالاتر بررسی کرده ایم را در نظر بگیرید و کد context.ChangeTracker.DetectChanges آن را حذف کنید و بار دیگر آن را فرا بخوانید و نتیجه را مشاهده کنید:

```
Before DetectChanges: Modified
After DetectChanges: Modified
```

این دفعه،EF از تغییرات صورت گرفته آگاه است،حال چه DetectChanges فراخوانده شود یا نشود.

اکنون متد DetectRelationshipChanges را ویرایش کرده و برنامه را اجرا کنید:

```
Before DetectChanges: will
After DetectChanges: will
```

این بار می‌بینیم که EF، تغییر رابطه را تشخیص داده و اصلاح رابطه را بدون فراخوانی DetectChanges انجام داده است.

نکته: زمانی که شما از پروکسی‌های ردیابی تغییر استفاده می‌کنید،احتیاجی به غیرفعال کردن تشخیص خودکار تغییرات نیست. DetectChanges برای همه اشیایی که تغییرات را به صورت بلادرنگ گزارش می‌دهند،فرآیند تشخیص تغییرات را انجام نمی‌دهد. بنابراین فعال سازی پروکسی‌های ردیابی تغییر،برای رسیدن به مزایای کارایی بالا در هنگام عدم استفاده از DetectChanges کافی است. در حقیقت زمانی که EF، یک پروکسی ردیابی پیدا می‌کند، از مقادیر خاصیت ها، عکس فوری نمی‌گیرد. همچنین DetectChanges این را نیز می‌داند که نباید تغییرات موجودیت هایی که عکسی از مقادیر اصلی آنها ندارد را اسکن کند.

تذکر: اگر شما موجودیت هایی داشته باشید که شامل انواع پیچیده(Complex Types) می‌شوند،EF هنوز هم از ردیابی تغییر عکس فوری، برای خواص موجود در نوع پیچیده استفاده می‌کند، و از این جهت لازم است کهEF، برای نمونه‌ی نوع پیچیده، پروکسی ایجاد نمی‌کند.شما هنوز هم، تشخیص خودکار تغییرات خواصی که مستقیما درون آن موجودیت(Entity) تعریف شده اند را دارید، ولی تغییرات رخ داده درون نوع پیچیده، فقط از طریق DetectChanges قابل تشخیص است.

چگونگی اطمینان از اینکه نمونه‌های جدید ، پروکسی‌ها را دریافت خواهند کرد

EF به صورت خودکار برای نتایج حاصل از کوئری هایی که شما اجرا می‌کنید، پروکسی‌ها را ایجاد می‌کند. با این حال اگر شما فقط از سازنده‌ی کلاس POCO خود برای ایجاد نمونه‌ی جدید استفاده کنید،دیگر پروکسی‌ها ایجاد نخواهند شد.بدین منظور برای دریافت پروکسی ها، شما باید از متد DbSet.Create برای دریافت نمونه‌های جدید آن موجودیت استفاده کنید.
نکته: اگر شما، پروکسی‌های ردیابی تغییر را برای موجودیتی از مدلتان فعال کرده باشید،هنوز هم می‌توانید،نمونه‌های فاقد پروکسی آن موجودیت را ایجاد و بیافزایید.خوشبختانه EF با موجودیت‌های پروکسی و غیر پروکسی در همان مجموعه(set) کار می‌کند.شما باید آگاه باشید که ردیابی خودکار تغییرات و یا اصلاح رابطه، برای نمونه هایی که پروکسی هایی ردیابی تغییر نیستند، قابل استفاده نیستند.داشتن مخلوطی از نمونه‌های پروکسی و غیر پروکسی در همان مجموعه، می‌تواند گیج کننده باشد.بنابر این عموما توصیه می‌شود که برای ایجاد نمونه‌های جدید از DbSet.Create استفاده کنید، تا همه‌ی موجودیت‌های موجود در مجموعه، پروکسی‌های ردیابی تغییر باشند.
متد CreateNewProxies را به برنامه‌ی خود اضافه کرده و آن را اجرا کنید.

```
private static void CreateNewProxies()
        {
            using (var context = new PhoneBookDbContext())
            {
                var phoneNumber = new PhoneNumber { Number = "987" };

                var davidPersonProxy = context.Persons.Create();
                davidPersonProxy.FirstName = "david";
                davidPersonProxy.PhoneNumbers.Add(phoneNumber);

                Console.WriteLine(phoneNumber.Person.FirstName);
            }
        }
```

خروجی مثال فوق david خواهد بود.همان طور که می‌بینید با استفاده از context.Persons.Create، نمونه‌ی ساخته شده، دیگر شی POCO نیست، بلکه davidPersonProxy، از جنس پروکسی ردیابی تغییر است و تغییرات آن به طور خودکار ردیابی شده و رابطه آن نیز به صورت خودکار اصلاح می‌شود.در اینجا نیز با افزودن phoneNumber به شماره تلفن‌های davidPersonProxy، به طور خودکار رابطه‌ی بین phoneNumber و davidPersonPeroxy بر قرار شده است. همان طور که می‌دانید این عملیات بدون استفاده از پروکسی‌های ردیابی تغییرات امکان پذیر نیست و موجب بروز خطا می‌شود.

ایجاد نمونه‌های پروکسی برای انواع مشتق شده

اورلود جنریک دیگری برای DbSet.Create وجود دارد که برای نمونه سازی کلاس‌های مشتق شده در مجموعه ما استفاده می‌شود .برای مثال، فراخوانی Create بر روی مجموعه‌ی Persons ،نمونه ای از کلاس Person را بر می‌گرداند.ولی ممکن است کلاس هایی در مجموعه‌ی Persons وجود داشته باشند، که از آن مشتق شده باشند، مانند Student. برای دریافت نمونه‌ی پروکسی Student، از اورلود جنریک Create استفاده می‌کنیم.

```
var newStudent = context.Persons.Create<Student>();
```

واکشی موجودیت‌ها بدون ردیابی تغییرات

تا به این جای کار باید متوجه شده باشید که ردیابی تغییرات، فرآیندی ساده و بدیهی نیست و مقداری سربار در کار است. در بعضی از بخش‌های برنامه تان، احتمالا داده‌ها را به صورت فقط خواندنی در اختیار کاربران قرار می‌دهید و چون اطلاعات هیچ وقت تغییر نمی‌کنند، شما می‌خواهید که سربار ناشی از ردیابی تغییرات را حذف کنید.
خوشبختانه EF شامل متد AsNoTracking است که می‌توان از آن برای اجرای کوئری‌های بدون ردیابی استفاده کرد.یک کوئری بدون ردیابی، یک کوئری ساده هست که نتایج آن توسط context برای تشخیص تغییرات ردیابی نخواهد شد.

متد PrintPersonsWithoutChangeTracking را به برنامه اضافه کنید و آن را اجرا کنید:

```
private static void PrintPersonsWithoutChangeTracking()
        {
            using (var context = new PhoneBookDbContext())
            {
                var persons = context.Persons.AsNoTracking().ToList();

                foreach (var person in persons)
                {
                    Console.WriteLine(person.FirstName);
                }

            }
        }
```

در مثال بالا از متد AsNoTracking برای گرفتن کوئری فاقد ردیابی استفاده کردیم تا محتویات مجموعه Persons را دریافت کنیم. در نهایت با یک حلقه‌ی foreach، نتایج را بر روی کنسول به نمایش در آوردیم.به دلیل اینکه، این یک کوئری بدون ردیابی هست، context دیگر تغییراتی که روی Persons رخ می‌دهد را ردیابی نمی‌کند.در نتیجه اگر شما یکی از خواص یکی از Persons را تغییردهید و SaveChanges را صدا بزنید، تغییرات به دیتابیس ارسال نمی‌شوند.

نکته: واکشی داده‌ها بدون ردیابی تغییرات،معمولا وقتی باعث افزایش قابل توجه کارایی می‌شود که بخواهیم تعداد خیلی زیادی داده را به صورت فقط خواندنی نمایش دهیم. اگر برنامه‌ی شما داده ای را تغییر می‌دهد و می‌خواهد آن را ذخیره کند، باید از AsNoTracking استفاده نکنید.

AsNoTracking یک متد الحاقی است، که در IQueryable<T> تعریف شده است، در نتیجه شما می‌توانید از آن، در کوئری‌های LINQ نیز استفاده کنید. شما می‌توانید از AsNoTracking، در انتهای DbSet ،در خط from کوئری استفاده کنید.

```
var query = from p in context.Persons.AsNoTracking()
                            where p.FirstName == "joe"
                            select p;
```

شما همچنین از AsNoTracking می‌توانید برای تبدیل یک کوئری LINQ موجود، به یک کوئری فاقد ردیابی استفاده کنید. این نکته را به یاد داشته باشید که فقط AsNoTracking بر روی کوئری، فرانخوانده شده است، بلکه متغیر query را با نتیجه‌ی حاصل از فراخوانی AsNoTracking بازنویسی(override) کرده است و این، از این جهت لازم است که AsNoTracking ،تغییری در کوئری ای که بر روی آن فراخوانده شده نمی‌دهد، بلکه یک کوئری جدید بر می‌گرداند.

```
var query = from p in context.Persons
                            where p.FirstName == "joe"
                            select p;
                query = query.AsNoTracking();
```

نکته: به دلیل اینکه AsNoTracking یک متد الحاقی است، شما احتیاج به افزودن فضای نام System.Data.Entity به فضاهای نام خود دارید.

منبع: ترجمه ای آزاد از کتاب Programming Entity Framework: DbContext

نظرات خوانندگان

نویسنده: امیر خلیلی
تاریخ: ۱۳۹۲/۰۸/۰۴ ۱۳:۳۸

یعنی با یکی از 2 روش گفته شده در بالا میتوان دیتابیس را مانیتور کرد و از تغییرات ایجاد شده در دیتابیس در همان لحظه با خبر شد ؟ مثلا ثبت نام یک کاربر جدید , یا ارسال یک نظر جدید و همان لحظه نمایش یک پیغام در صفحه ادمین؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۰۴ ۱۳:۴۶

خیر. در ORMها کلا ردیابی منظور ردیابی تغییرات انجام شده در اشیایی است که در حال کار با آن‌ها هستیم آن هم در طی یک Context موجود. مثلا در یک Context باز شده و فعال، یک شیء اضافه می‌شود. دو خاصیت شیء ایی دیگر ویرایش می‌شوند. دو شیء دیگر نیز حذف خواهند شد. اینجا است که ORM باید بتواند این موارد و تغییرات را ردیابی کرده و سپس SQL صحیح و بهینه‌ای را جهت اعمال بر روی بانک اطلاعاتی تولید کند.

نویسنده: امیر خلیلی
تاریخ: ۱۳۹۲/۰۸/۰۴ ۱۳:۵۳

خیلی ممنون از جوابتون
اگه امکان داره لطف بفرمایین با یک راهنمایی کوچک که برای مانیتور دیتابیس و اون هدفی که در بالا گفتم از چه روشی میتوان استفاده کرد ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۰۴ ۱۳:۵۹

دوره « معرفی SignalR و ارتباطات بلادرنگ » در سایت می‌تونه شروع خوبی باشه.

نویسنده: سوین
تاریخ: ۱۳۹۲/۰۹/۰۹ ۰:۴۴

با سلام
من قبلا در EF 4 برای ذخیره اطلاعات با استفاده از دیتاگرید در WPF App می‌اومدم یه Context ایجاد می‌کردم و اطلاعات رو از جدول به صورت IQueryable به ItemSource دیتاگراید بایند می‌کردم و بعد از تغییر اطلاعات در انتها با یه SaveChange تغییرات رو تو دیتابیس ذخیره می‌شد اما الان در EF 6 این خطا رو می‌ده

```
Data binding directly to a store query (DbSet, DbQuery, DbSqlQuery) is not supported. Instead populate a DbSet with data, for example by calling Load on the DbSet, and then bind to local data. For WPF bind to DbSet.Local. For WinForms bind to DbSet.Local.ToBindingList().
```

ممنون میشم راهنماییم کنید .

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۹/۰۹ ۰:۵۸

استفاده از خاصیت Local در Entity Framework
مدیریت تغییرات گریدی از اطلاعات به کمک استفاده از الگوی واحد کار مشترک بین ViewModel و لایه سرویس

نویسنده: مجتبی فخاری
تاریخ: ۱۳۹۲/۱۲/۰۸ ۱۷:۸

با این تفاسیر الان ما باید خاصیت‌های کلاس هامون، مثلا Id رو هم به صورت virtual تعریف کنیم؟ یا لزومی نداره؟

نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۲/۱۲/۰۹ ۸:۳۳

به شخصه من این کار را انجام میدهم. ولی یادم هست که در یک پروژه و در یک سناریوی خاص Entity framework یک استثنا صادر می‌کرد که با جست و جو در اینترنت، یکی از اعضای توسعه دهنده‌ی تیم Entity framework گفته بود که در این سناریو، Entity framework توانایی کار با تمام اعضای virtual را ندارد.
البته این موضوع به به نسخه‌ی 4.3 بر میگرده و احتمالش هست که اشکالش در نسخه‌های بعد رفع شده باشد.
از نظر شخصی خودم در پروژه هاتون به خصوص پروژه‌های ویندوزی به عنوان یک best practice همه‌ی اعضا را virtual تعریف کنید مگر اینکه به مشکل بر بخورید.