عنوان: **مدیریت Join در NHibernate 3.0**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۰/۱۷ ۱۴:۲۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: NHibernate

مباحث eager fetching/loading (واکشی حریصانه) و lazy loading/fetching (واکشی در صورت نیاز، با تاخیر، تنبل) جزو نکات کلیدی کار با ORM های پیشرفته بوده و در صورت عدم اطلاع از آن‌ها و یا استفاده‌ی ناصحیح از هر کدام، باید منتظر از کار افتادن زود هنگام سیستم در زیر بار چند کاربر همزمان بود. به همین جهت تصور اینکه "با استفاده از ORMs دیگر از فراگیری SQL راحت شدیم!" یا اینکه "به من چه که پشت صحنه چه اتفاقی می‌افته!" بسی مهلک و نادرست است!
در ادامه به تفصیل به این موضوع پرداخته خواهد شد.

ابزار مورد نیاز

در این مطلب از برنامه‌ی NHProf استفاده خواهد شد.
اگر مطالب NHibernate این سایت را دنبال کرده باشید، در مورد لاگ کردن SQL تولیدی به اندازه‌ی کافی توضیح داده شده یا حتی یک ماژول جمع و جور هم برای مصارف دم دستی نوشته شده است . این موارد شاید این ایده را به همراه داشته باشند که چقدر خوب می‌شد یک برنامه‌ی جامع‌تر برای این نوع بررسی‌ها تهیه می‌شد. حداقل SQL نهایی فرمت می‌شد (یعنی برنامه باید مجهز به یک SQL Parser تمام عیار باشد که کار چند ماهی هست ...؛ با توجه به اینکه مثلا NHibernate از افزونه‌های SQL ویژه بانک‌های اطلاعاتی مختلف هم پشتیبانی می‌کند، مثلا T-SQL مایکروسافت با یک سری ریزه کاری‌های منحصر به MySQL متفاوت است)، یا پس از فرمت شدن، syntax highlighting به آن اضافه می‌شد، در ادامه مشخص می‌کرد کدام کوئری‌ها سنگین‌تر هستند، کدامیک نشانه‌ی عدم استفاده‌ی صحیح از ORM مورد استفاده است، چه مشکلی دارد و از این موارد.
خوشبختانه این ایده‌ها یا آرزوها با برنامه‌ی NHProf محقق شده است. این برنامه برای استفاده‌ی یک ماه اول آن رایگان است (آدرس ایمیل خود را وارد کنید تا یک فایل مجوز رایگان یک ماهه برای شما ارسال گردد) و پس از یک ماه، باید حداقل 300 دلار هزینه کنید.

واکشی حریصانه و غیرحریصانه چیست؟

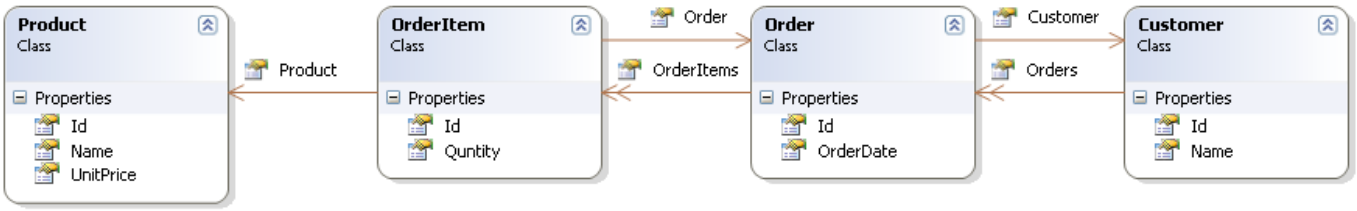
رفتار یک ORM جهت تعیین اینکه آیا نیاز است برای دریافت اطلاعات بین جداول Join صورت گیرد یا خیر، واکشی حریصانه و غیرحریصانه را مشخص می‌سازد.
در حالت واکشی حریصانه به ORM خواهیم گفت که لطفا جهت دریافت اطلاعات فیلدهای جداول مختلف، از همان ابتدای کار در پشت صحنه، Join های لازم را تدارک ببین. در حالت واکشی غیرحریصانه به ORM خواهیم گفت به هیچ عنوان حق نداری Join ایی را تشکیل دهی. هر زمانی که نیاز به اطلاعات فیلدی از جدولی دیگر بود باید به صورت مستقیم به آن مراجعه کرده و آن مقدار را دریافت کنی.
به صورت خلاصه برنامه نویس در حین کار با ORM های پیشرفته نیازی نیست Join بنویسد. تنها باید ORM را طوری تنظیم کند که آیا اینکار را حتما خودش در پشت صحنه انجام دهد (واکشی حریصانه)، یا اینکه خیر، به هیچ عنوان SQL های تولیدی در پشت صحنه نباید حاوی Join باشند (lazy loading).

چگونه واکشی حریصانه و غیرحریصانه را در NHibernate 3.0 تنظیم کنیم؟

در NHibernate اگر تنظیم خاصی را تدارک ندیده و خواص جداول خود را به صورت virtual معرفی کرده باشید، تنظیم پیش فرض دریافت اطلاعات همان lazy loading است. به مثالی در این زمینه توجه بفرمائید:

مدل برنامه:
مدل برنامه همان مثال کلاسیک مشتری و سفارشات او می‌باشد. هر مشتری چندین سفارش می‌تواند داشته باشد. هر سفارش به

یک مشتری وابسته است. هر سفارش نیز از چندین قلم جنس تشکیل شده است. در این خرید، هر جنس نیز به یک سفارش وابسته است.




```
using System.Collections.Generic;
namespace CustomerOrdersSample.Domain
{
   public class Customer
   {
       public virtual int Id { get; set; }
       public virtual string Name { get; set; }
       public virtual IList<Order> Orders { get; set; }
   }
}
```

```
using System;
using System.Collections.Generic;
namespace CustomerOrdersSample.Domain
{
   public class Order
   {
       public virtual int Id { get; set; }
       public virtual DateTime OrderDate { set; get; }
       public virtual Customer Customer { get; set; }
       public virtual IList<OrderItem> OrderItems { set; get; }
   }
}
```

```
namespace CustomerOrdersSample.Domain
{
   public class OrderItem
   {
       public virtual int Id { get; set; }
       public virtual Product Product { get; set; }
       public virtual int Quntity { get; set; }
       public virtual Order Order { set; get; }
   }
}
```

```
namespace CustomerOrdersSample.Domain
{
   public class Product
   {
       public virtual int Id { set; get; }
       public virtual string Name { get; set; }
       public virtual decimal UnitPrice { get; set; }
   }
}
```

که جداول متناظر با آن به صورت زیر خواهند بود:

```
create table Customers (
      CustomerId INT IDENTITY NOT NULL,
      Name NVARCHAR(255) null,
      primary key (CustomerId)
   )

   create table Orders (
      OrderId INT IDENTITY NOT NULL,
      OrderDate DATETIME null,
      CustomerId INT null,
      primary key (OrderId)
   )

   create table OrderItems (
      OrderItemId INT IDENTITY NOT NULL,
      Quntity INT null,
      ProductId INT null,
      OrderId INT null,
      primary key (OrderItemId)
   )

   create table Products (
      ProductId INT IDENTITY NOT NULL,
      Name NVARCHAR(255) null,
      UnitPrice NUMERIC(19,5) null,
      primary key (ProductId)
   )

   alter table Orders
       add constraint fk_Customer_Order
       foreign key (CustomerId)
       references Customers

   alter table OrderItems
       add constraint fk_Product_OrderItem
       foreign key (ProductId)
       references Products

   alter table OrderItems
       add constraint fk_Order_OrderItem
       foreign key (OrderId)
       references Orders
```

همچنین یک سری اطلاعات آزمایشی زیر را هم در نظر بگیرید: (بانک اطلاعاتی انتخاب شده SQL CE است)

```
SET IDENTITY_INSERT [Customers] ON;
GO
INSERT INTO [Customers] ([CustomerId],[Name]) VALUES (1,N'Customer1');
GO
SET IDENTITY_INSERT [Customers] OFF;
GO
SET IDENTITY_INSERT [Products] ON;
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES (1,N'Product1',1000.00000);
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES (2,N'Product2',2000.00000);
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES (3,N'Product3',3000.00000);
GO
SET IDENTITY_INSERT [Products] OFF;
GO
SET IDENTITY_INSERT [Orders] ON;
GO
INSERT INTO [Orders] ([OrderId],[OrderDate],[CustomerId]) VALUES (1,{ts '2011-01-07 11:25:20.000'},1);
GO
SET IDENTITY_INSERT [Orders] OFF;
GO
SET IDENTITY_INSERT [OrderItems] ON;
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (1,10,1,1);
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (2,5,2,1);
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (3,20,3,1);
GO
SET IDENTITY_INSERT [OrderItems] OFF;
```

```
GO
```

دریافت اطلاعات :

می‌خواهیم نام کلیه محصولات خریداری شده توسط مشتری‌ها را به همراه نام مشتری و زمان خرید مربوطه، نمایش دهیم (دریافت اطلاعات از 4 جدول بدون join نویسی):

```
var list = session.QueryOver<Customer>().List();

foreach (var customer in list)
{
      foreach (var order in customer.Orders)
      {
          foreach (var orderItem in order.OrderItems)
          {
                Console.WriteLine("{0}:{1}:{2}", customer.Name, order.OrderDate,
orderItem.Product.Name);
           }
       }
}
```

خروجی به صورت زیر خواهد بود:

```
Customer1:2011/01/07 11:25:20 :Product1
Customer1:2011/01/07 11:25:20 :Product2
Customer1:2011/01/07 11:25:20 :Product3
```

اما بهتر است نگاهی هم به پشت صحنه عملیات داشته باشیم:

همانطور که مشاهده می‌کنید در اینجا اطلاعات از 4 جدول مختلف دریافت می‌شوند اما ما Join ایی را ننوشته‌ایم. ORM هرجایی که به اطلاعات فیلدهای جداول دیگر نیاز داشته، به صورت مستقیم به آن جدول مراجعه کرده و یک کوئری، حاصل این عملیات خواهد بود (مطابق تصویر جمعا 6 کوئری در پشت صحنه برای نمایش سه سطر خروجی فوق اجرا شده است).

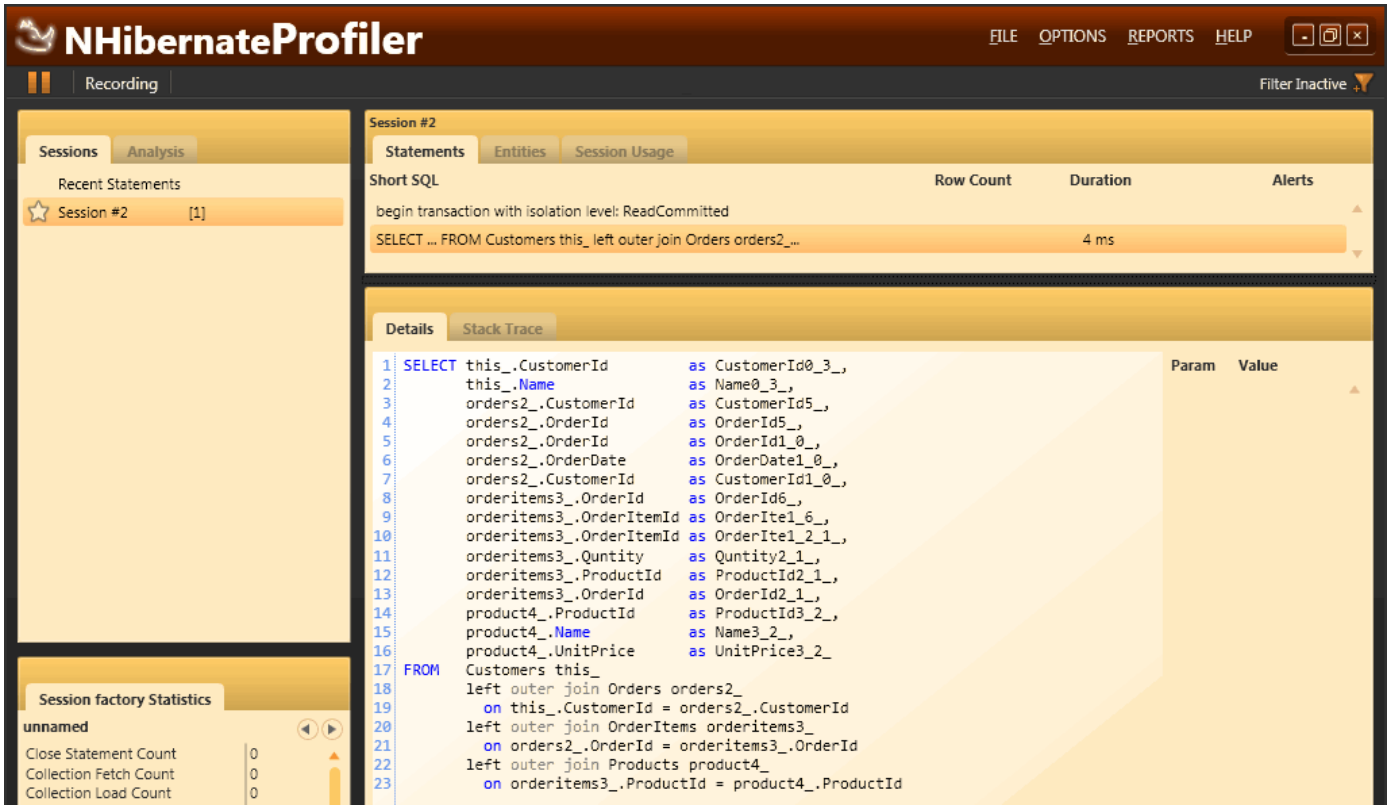
این حالت فقط و فقط با تعداد رکورد کم بهینه است (و به همین دلیل هم تدارک دیده شده است). بنابراین اگر برای مثال قصد نمایش اطلاعات حاصل از 4 جدول فوق را در یک گرید داشته باشیم، بسته به تعداد رکوردها و تعداد کاربران همزمان برنامه (خصوصا در برنامه‌های تحت وب)، بانک اطلاعاتی باید بتواند هزاران هزار کوئری رسیده حاصل از lazy loading را پردازش کند و این یعنی مصرف بیش از حد منابع (IO بالا، مصرف حافظه بالا) به همراه بالا رفتن CPU usage و از کار افتادن زود هنگام سیستم.
کسانی که پیش از این با SQL نویسی خو گرفته‌اند احتمالا الان منابع موجود را در مورد نحوه‌ی نوشتن Join در NHibernate زیر و رو خواهند کرد؛ زیرا پیش از این آموخته‌اند که برای دریافت اطلاعات از دو یا چند جدول مرتبط باید Join نوشت. اما همانطور که پیشتر نیز عنوان شد، اگر با جزئیات کار با NHibernate آشنا شویم، نیازی به Join نویسی نخواهیم داشت. اینکار را خود ORM در پشت صحنه باید و می‌تواند مدیریت کند. اما چگونه؟
در NHibernate 3.0 با معرفی QueryOver که جایگزینی از نوع strongly typed همان ICriteria API قدیمی است، یا با معرفی Query که همان LINQ to NHibernate می‌باشد، متدی به نام Fetch نیز تدارک دیده شده است که استراتژی‌های lazy loading و eager loading را به سادگی توسط آن می‌توان مشخص نمود.

مثال: دریافت اطلاعات با استفاده از QueryOver

```
var list = session
                  .QueryOver<Customer>()
                  .Fetch(c => c.Orders).Eager
                  .Fetch(c => c.Orders.First().OrderItems).Eager
                  .Fetch(c => c.Orders.First().OrderItems.First().Product).Eager
                  .List();

foreach (var customer in list)
{
     foreach (var order in customer.Orders)
     {
           foreach (var orderItem in order.OrderItems)
           {
               Console.WriteLine("{0}:{1}:{2}", customer.Name, order.OrderDate,
orderItem.Product.Name);
            }
      }
}
```

پشت صحنه:



اینبار فقط یک کوئری حاصل عملیات بوده و join ها به صورت خودکار با توجه به متدهای Fetch ذکر شده که حالت eager loading آن‌ها صریحا مشخص شده است، تشکیل شده‌اند (6 بار رفت و برگشت به بانک اطلاعاتی به یکبار تقلیل یافت).

نکته 1: نتایج تکراری

اگر حاصل join آخر را نمایش دهیم، نتایجی تکراری خواهیم داشت که مربوط است به مقدار دهی customer با سه وهله از شیء مربوطه تا بتواند واکشی حریصانه‌ی مجموعه اشیاء فرزند آن‌را نیز پوشش دهد. برای رفع این مشکل یک سطر TransformUsing باید اضافه شود:

```
...
.TransformUsing(NHibernate.Transform.Transformers.DistinctRootEntity)
.List();
```

دریافت اطلاعات با استفاده از LINQ to NHibernate3.0

برای اینکه بتوان متدهای Fetch ذکر شده را به LINQ to NHibernate 3.0 اعمال نمود، ذکر فضای نام NHibernate.Linq ضروری است. پس از آن خواهیم داشت:

```
var list = session
```

```
.Query
```

اینبار از FetchMany، سپس ThenFetchMany (برای واکشی حریصانه مجموعه‌های فرزند) و در آخر از ThenFetch استفاده خواهد شد.

همانطور که ملاحظه می‌کنید حاصل این کوئری، با کوئری قبلی ذکر شده یکسان است. هر دو، اطلاعات مورد نیاز از دو جدول مختلف را نمایش می‌دهند. اما یکی در پشت صحنه شامل چندین و چند کوئری برای دریافت اطلاعات است، اما دیگری تنها از یک کوئری Join دار تشکیل شده است.

نکته 2: خطاهای ممکن

ممکن است حین تعریف متدهای Fetch در زمان اجرا به خطاهای Antlr.Runtime.MismatchedTreeNodeException و یا Specified method is not supported و یا موارد مشابهی برخورد نمائید. تنها کاری که باید انجام داد جابجا کردن مکان بکارگیری extension methods است. برای مثال متد Fetch باید پس از Where در حالت استفاده از LINQ ذکر شود و نه قبل از آن.

نظرات خوانندگان

نویسنده: مهدی پایروند
تاریخ: ۱۳۸۹/۱۰/۱۹ ۲۳:۱۳:۲۹

سلام، مطلب جالبی بود، البته تا اونجایی که من میدونم برای مثال در بانک اطلاعاتی اوراکل اگر از جوین بیشتر از 3 تا استفاده کنید سرعت دریافت اطلاعات به شدت پایین میاد

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۰/۲۰ ۰۰:۲۷:۳۵

برای دریافت مجوز یک ماهه برنامه‌ی NHProf به همان صفحه http://nhprof.com/Trial مراجعه کرده و ایمیل خود را وارد کنید.