

目录

NHibernate 之旅系列文章导航	2
宣传语.....	2
导游.....	2
环境优先.....	2
休息接待区.....	2
旅途站点路线.....	2
NHibernate 之旅(1): 开篇有益	4
NHibernate 开篇有益	4
NHibernate 是什么	5
NHibernate 的架构	5
NHibernate 资源	6
欢迎加入 NHibernate 中文社区	7
NHibernate 之旅(2): 第一个 NHibernate 程序	8
开始使用 NHibernate	8
1.获取 NHibernate	9
2.建立数据库表	9
3.创建 C#类库项目	10
4.编写 DomainModel 层	11
5.编写数据访问层	13
6.编写数据访问层的测试	15
结语	17
NHibernate 之旅(3): 探索查询之 NHibernate 查询语言(HQL)	18
NHibernate 中的查询方法	18
NHibernate 查询语言(HQL)	19
实例分析	22
结语	24
NHibernate 之旅(4): 探索查询之条件查询(Criteria Query)	25
NHibernate 中的查询方法	25
条件查询(Criteria Query)	25
根据示例查询(Query By Example)	27
实例分析	28
结语	29
NHibernate 之旅(5): 探索 Insert, Update, Delete 操作	30
操作数据概述	30
1.新建对象	30
2.删除对象	31
3.更新对象	32
4.保存更新对象	32
结语	34
NHibernate 之旅(6): 探索 NHibernate 中的事务	35
事务概述	35

1.新建对象.....	36
2.删除对象.....	39
3.更新对象.....	39
4.保存更新对象.....	40
结语.....	41
NHibernate 之旅(7): 初探 NHibernate 中的并发控制.....	42
什么是并发控制?	42
悲观并发控制(Pessimistic Concurrency).....	42
乐观并发控制(Optimistic Concurrency).....	42
NHibernate 支持乐观并发控制.....	43
实例分析.....	44
结语.....	47
NHibernate 之旅(8): 巧用组件之依赖对象.....	48
引入.....	48
方案 1: 直接添加.....	48
方案 2: 巧用组件.....	48
实例分析.....	49
结语.....	53
NHibernate 之旅(9): 探索父子关系(一对多关系).....	54
引入.....	54
NHibernate 中的集合类型.....	54
建立父子关系.....	55
父子关联映射.....	57
结语.....	61
NHibernate 之旅(10): 探索父子(一对多)关联查询.....	62
关联查询引入.....	62
一对多关联查询.....	62
结语.....	66
NHibernate 之旅(11): 探索多对多关系及其关联查询.....	67
多对多关系引入.....	67
多对多映射关系.....	68
多对多关联查询.....	71
结语.....	74
NHibernate 之旅(12): 初探延迟加载机制.....	75
引入.....	75
延迟加载(Lazy Loading).....	76
实例分析.....	76
1.一对多关系实例.....	76
2.多对多关系实例.....	78
结语.....	82
NHibernate 之旅(13): 初探立即加载机制.....	83
引入.....	83
立即加载.....	83
实例分析.....	84

1.一对多关系实例.....	84
2.多对多关系实例.....	86
结语.....	91
NHibernate 之旅(14): 探索 NHibernate 中使用视图.....	92
引入.....	92
1.持久化类.....	93
2.映射文件.....	93
3.测试.....	94
结语.....	95
NHibernate 之旅(15): 探索 NHibernate 中使用存储过程(上).....	96
引入.....	96
使用 MyGeneration 生成存储过程.....	96
实例分析.....	99
结语.....	102
NHibernate 之旅(16): 探索 NHibernate 中使用存储过程(中).....	103
引入.....	103
实例分析.....	103
结语.....	108
NHibernate 之旅(17): 探索 NHibernate 中使用存储过程(下).....	109
引入.....	109
实例分析.....	109
拾遗.....	113
结语.....	114
NHibernate 之旅(18): 初探代码生成工具使用.....	115
引入.....	115
代码生成工具.....	116
结语.....	124
NHibernate 之旅(19): 初探 SchemaExport 工具使用.....	125
引入.....	125
SchemaExport 工具.....	125
SchemaUpdate 工具.....	126
实例分析.....	126
结语.....	131
NHibernate 之旅(20): 再探 SchemaExport 工具使用.....	132
引入.....	132
实例分析.....	132
1.表及其约束.....	132
2.存储过程、视图.....	137
NHibernate 之旅(21): 探索对象状态.....	140
引入.....	140
对象状态.....	140
对象状态转换.....	141
结语.....	144

NHibernate 之旅系列文章导航

宣传语

NHibernate、NHibernate 教程、NHibernate 入门、NHibernate 下载、NHibernate 教程中文版、NHibernate 实例、NHibernate2.0、NHibernate2.0 教程、NHibernate 之旅、NHibernate 工具

导游

NHibernate 是把 Java 的 Hibernate 核心部分移植到 Microsoft .NET Framework 上。它是一个对象关系映射工具，其目标是把 .NET 对象持久化到关系数据库。

NHibernate 在 2008 年 8 月 31 日发布了 NHibernate2.0 版本，代表 NHibernate 又向前走了一步，我相信 NHibernate 将会越来越强大。唯一的遗憾是现在 NHibernate 对泛型的支持有点不足，源于 Java 中的泛型是编译时“擦拭法”泛型不是真正的泛型，如果 NHibernate 添加上自己独特的功能——泛型，那么更为强大了很多。

这个 NHibernate 之旅系列带你来到 NHibernate 的世界。一步一步看清 NHibernate 中的种种细节。

环境优先

这次 NHibernate2.0 系列之旅使用 Microsoft Visual Studio 2008 SP1、SQL Server 2008 Express、NHibernate2.0 最新环境，非常舒适。不过你可以到[这里](#)获得 NHibernate 最新版本，[这里](#)获得 NHibernate Contrib 最新版本。

休息接待区

欢迎加入 [NHibernate 中文社区](#)！在讨论中寻找乐趣！在问题中寻找答案！

请转向：<http://space.cnblogs.com/group/NHibernate>！全程接待！期待你的 [NHibernate 中文社区](#)之旅！

旅途站点路线

第一站：鸟瞰 NHibernate

[NHibernate 之旅\(1\): 开篇有益](#)

第二站：接触 NHibernate

[NHibernate 之旅\(2\): 第一个 NHibernate 程序](#)

第三站：数据在我手中

[NHibernate 之旅\(3\): 探索查询之 NHibernate 查询语言\(HQL\)](#)

[NHibernate 之旅\(4\): 探索查询之条件查询\(Criteria Query\)](#)

[NHibernate 之旅\(5\): 探索 Insert, Update, Delete 操作](#)

第四站：控制你的全部

[NHibernate 之旅\(6\): 探索 NHibernate 中的事务](#)

[NHibernate 之旅\(7\): 初探 NHibernate 中的并发控制](#)

观光站：实用技巧补偿

[NHibernate 之旅\(8\): 巧用组件之依赖对象](#)

第五站：关系如此复杂

[NHibernate 之旅\(9\): 探索父子关系\(一对多关系\)](#)

[NHibernate 之旅\(10\): 探索父子\(一对多\)关联查询](#)

[NHibernate 之旅\(11\): 探索多对多关系及其关联查询](#)

第六站：我来加载你

[NHibernate 之旅\(12\): 初探延迟加载机制](#)

[NHibernate 之旅\(13\): 初探立即加载机制](#)

第七站：数据的镜子

[NHibernate 之旅\(14\): 探索 NHibernate 中使用视图\(new!\)](#)

[NHibernate 之旅\(15\): 探索 NHibernate 中使用存储过程\(上\)\(new!\)](#)

[NHibernate 之旅\(16\): 探索 NHibernate 中使用存储过程\(中\)\(new!\)](#)

[NHibernate 之旅\(17\): 探索 NHibernate 中使用存储过程\(下\)\(new!\)](#)

第八站：转载请注明

[NHibernate 之旅\(18\): 初探代码生成工具使用\(new!\)](#)

[NHibernate 之旅\(19\): 初探 SchemaExport 工具使用\(new!\)](#)

[NHibernate 之旅\(20\): 再探 SchemaExport 工具使用\(new!\)](#)

下一站：停靠在哪儿

[NHibernate 之旅\(21\)](#): 探索对象状态(new!)

NHibernate 之旅(22): 探索离线查询

NHibernate 之旅(23): 探索级联操作

NHibernate 之旅(24): 探索一级缓存

NHibernate 之旅(25): 探索二级缓存

.....期待更新.....

最终站：旅途更新中

NHibernate 之旅(1): 开篇有益

本节内容

- NHibernate 是什么
- NHibernate 的架构
- NHibernate 资源
- 欢迎加入 [NHibernate 中文社区](#)

NHibernate 开篇有益

学习 NHibernate 有一段时间了，打算做个阶段性总结，就萌生了这个系列，这个系列参考 NHibernate 官方文档和 Steve Bohlen 的 NHibernate 之夏视频教程。作为开篇，首先了解多少人在使用 NHibernate，先搞清楚 NHibernate 是什么？学习 NHibernate 的一些资源。也欢迎大家加入 [NHibernate 中文社区](#)。

这个系列我使用 NHibernate 官方 2008 年 9 月 29 日最新发布的 NHibernate-2.0.1.GA 版本。开发环境是 Microsoft Visual Studio 2008 SP1、SQL Server 2008 Express、TestDriven.NET。你可以到[这里](#)下载获得 NHibernate 最新版本。到[这里](#)下载获得 NHibernate Contrib 最新版本。2.0 版比 1.2 版本添加了很多特性和改进。可惜 2.0 版本没有发布 LINQ for NHibernate，不过在接下来的 2.1 版本会发布 LINQ for NHibernate，如果你现在很想尝鲜 LINQ for NHibernate，你可以在[这里](#)找到社区版的 NHibernate.Linq.dll。

NHibernate 是什么

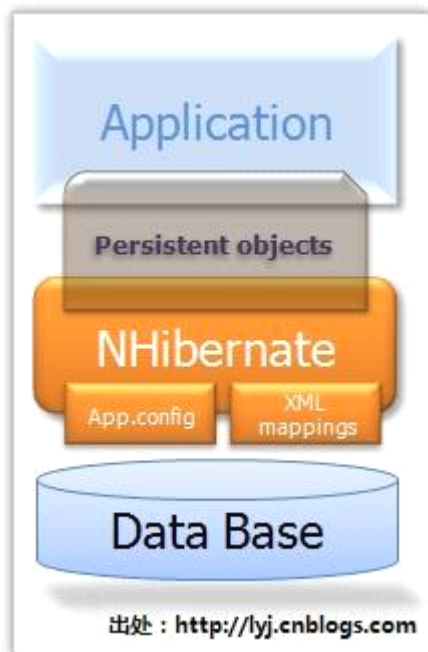
NHibernate 是一个面向 .NET 环境的对象/关系数据库映射工具。对象关系映射(O/R Mapping, Object Relational Mapping)表示一种技术, 用来把对象模型表示的对象映射到基于 SQL 的关系模型数据结构中去。

NHibernate 不仅仅管理 .NET 类到数据库表的映射 (包括 .NET 数据类型到 SQL 数据类型的映射), 还提供数据查询和获取数据的方法, 大幅度减少我们开发时人工使用 SQL 和 ADO.NET 处理数据的时间。NHibernate 的目标是对于开发者通常的数据持久化相关的编程任务, 解放其中的 95%。请记住 NHibernate 作为数据库访问层, 是与你的程序紧密集成的。

NHibernate 的架构

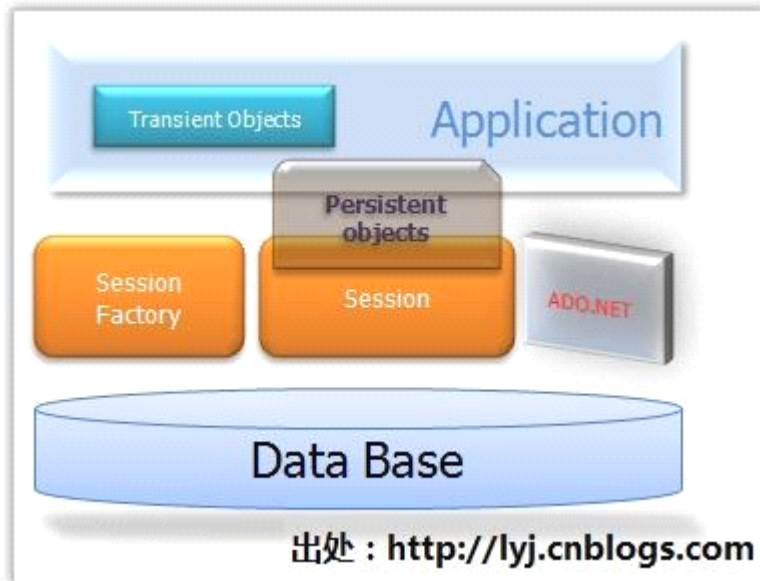
你知道 NHibernate 到底什么样子? 下面我摘取官方文档中的三幅不同的结构图稍做说明。

第一幅图: NHibernate 体系结构非常抽象的概览

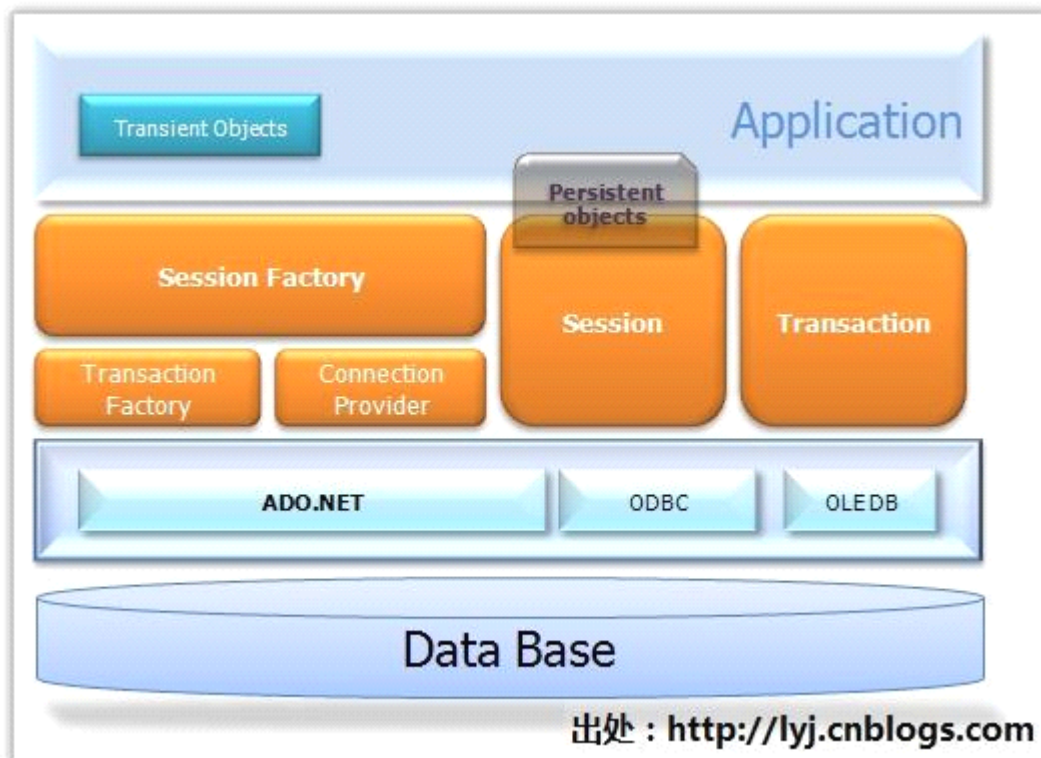


这幅图展示了 NHibernate 在数据库和应用程序之间提供了一个持久层。

第一幅图好像非常简单? 其实 NHibernate 是比较复杂的。我们了解两种极端情况, 轻量级和重量级架构。再来第二幅图: 轻量级体系, 应用程序自己提供 ADO.NET 连接, 并且自行管理事务。



最后一张图：重量级体系：所有的底层 ADO.NET API 都被抽象了。



NHibernate 资源

NHibernate 资源现在已经比较多了，但是大部分都是英文了，这里我仅仅挑选几个站点。

NHibernate 官方主页: <http://www.nhibernate.org/>(英文)

NHibernate 社区: <http://www.nhforge.org/>(英文)

NHibernate 参考文档 2.0.0: <http://nhforge.org/doc/nh/en/>(英文)

NHibernate 之夏系列录像教程: <http://www.summerofnhibernate.com/>(英文)

欢迎加入 [NHibernate 中文社区](#)

<http://space.cnblogs.com/group/NHibernate>

为什么叫做 NHibernate 中文社区呢? 原因很简单, 体现本地化。现在关于 NHibernate 很多的资料都是英文资料, 中文资料少的可怜了也不是很完整, 我们努力建立在这个小组建立起来属于大家的 [NHibernate 中文社区](#), 在这里一起讨论 NHibernate、学习 NHibernate。

如果你使用 NHibernate, 学习 NHibernate, 欢迎加入这个小组, 一起讨论 NHibernate、学习 NHibernate, 一起建立 [NHibernate 中文社区](#)。

NHibernate 之旅(2): 第一个 NHibernate 程序

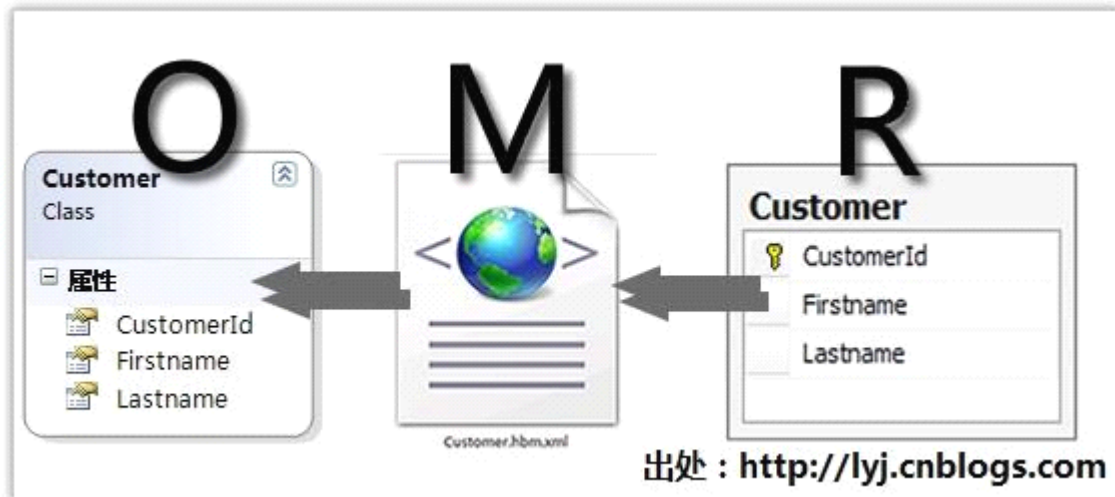
本节内容

- 开始使用 NHibernate
- 1.获取 NHibernate
- 2.建立数据库表
- 3.创建 C#类库项目
- 4.编写 DomainModel 层
 - 4-1.编写持久化类
 - 4-2.编写映射文件
- 5.编写数据访问层
 - 5-1.辅助类
 - 5-2.编写操作
- 6.编写数据访问层的测试
 - 6-1.配置 NHibernate
 - 6-2.测试
- 结语

开始使用 NHibernate

我们亲自动手，来一步一步搭建一个 NHibernate 程序来，我以一个实际场景电子交易程序来模拟，客户/订单/产品的经典组合。由于是第一次使用 NHibernate，所以我们的目的是映射一张表并完成使用 NHibernate 来读取数据，下面的一幅图片给了我们第一印象。我们按照基本开发软件思想的流程一步一步完成。

我使用的开发环境: Microsoft Visual Studio 2008 SP1、SQL Server 2008 Express、NHibernate 2.0 最新版。



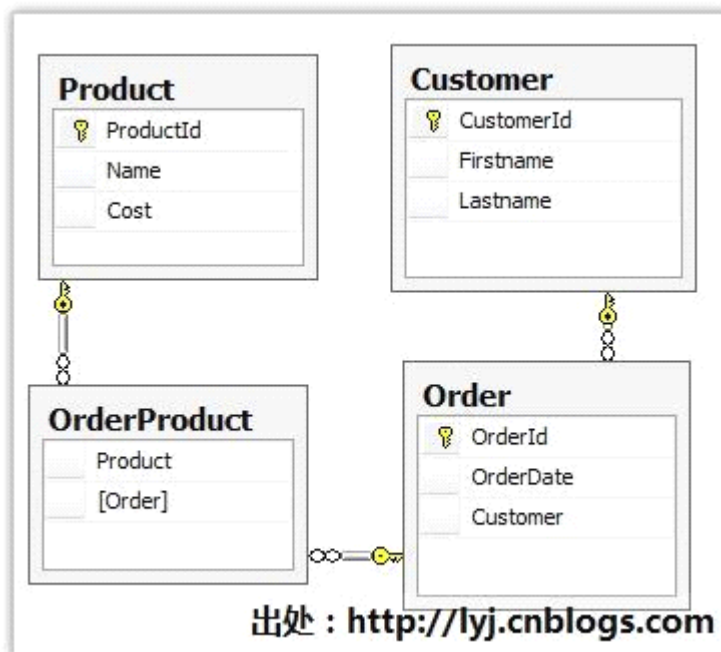
1. 获取 NHibernate

使用官方 2008 年 9 月 29 日最新发布的 NHibernate-2.0.1.GA 版本。你可以到这里[下载](#)获得 NHibernate 最新版本。到这里[下载](#)获得 NHibernate Contrib 最新版本。

2. 建立数据库表

打开 SQL Server Management Studio，新建一个新的数据库 NHibernateSample，创建四个表：分别为客户表、订单表、订单产品表、产品表。

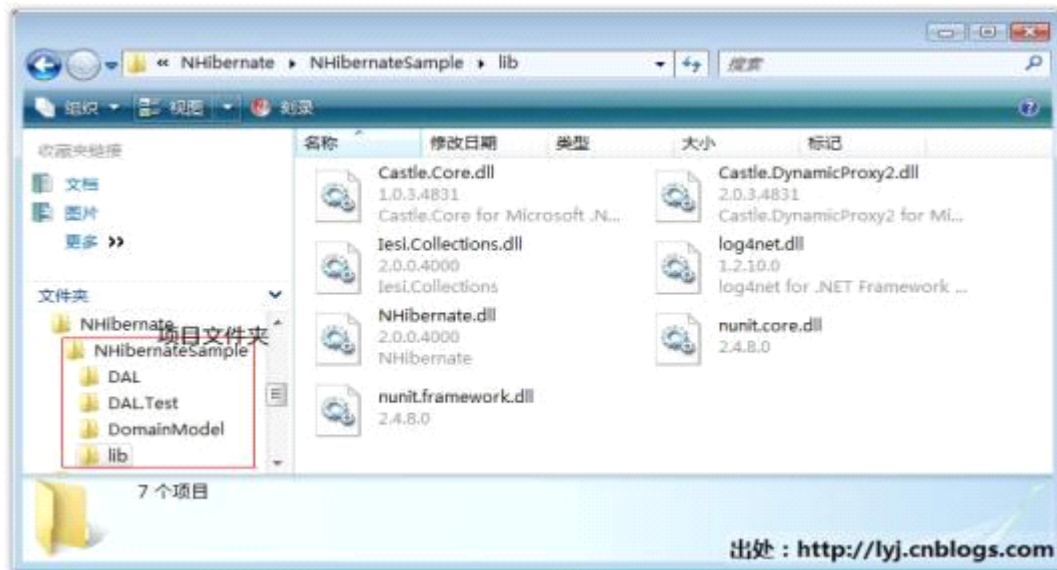
注：源码中附有创建数据库脚本。



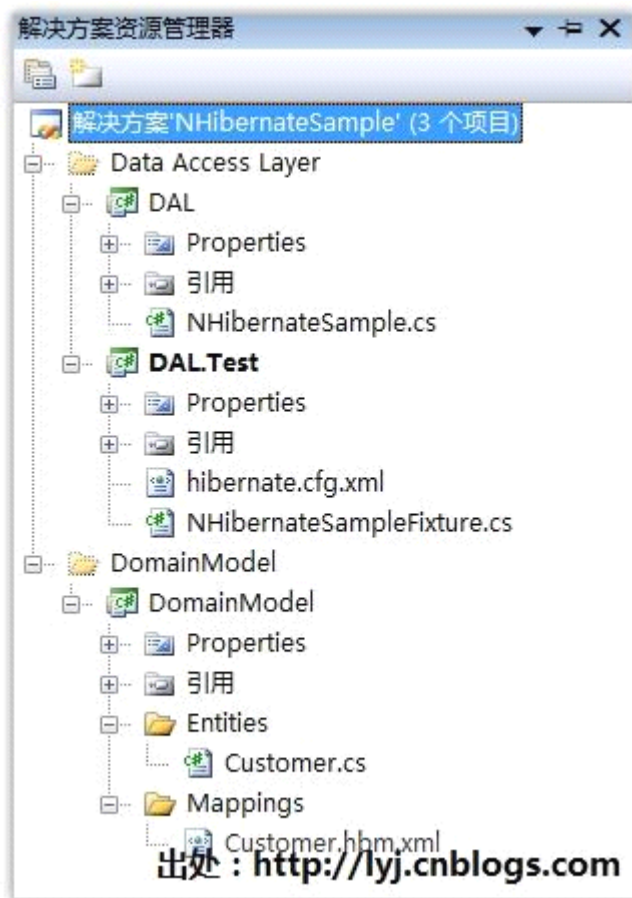
3.创建 C#类库项目

注意为什么创建 C#类库项目呢？在现在软件设计中，大多数都是采用多层架构来设计，**比较经典**的三层架构（页面表示层，业务逻辑层，数据访问层）通常而言业务逻辑层和数据访问层都是使用类库设计，页面表示层用 Web 应用程序设计，它引用业务逻辑层和数据访问层类库 DLL 程序集。

使用 VS2008 创建 C#类库的项目，命名为 NHibernateSample。打开项目文件夹，在其项目文件目录上新建 lib 文件夹，把下载 NHibernate 相关程序集文件拷贝到 lib 文件夹下。如下图：



创建项目，结构如下：



- DomainModel(域模型): 用于持久化类和 O/R Mapping 操作
- DAL(Data Access Layer 数据访问层): 定义对象的 CRUD 操作
- DAL.Test(数据访问层测试): 对数据访问层的测试, 这里我使用 Nunit 单元测试框架

项目引用

- DomainModel: 引用 Iesi.Collections.dll 程序集 (Set 集合在这个程序集中)
- DAL: 引用 NHibernate.dll 和 Iesi.Collections.dll 程序集, DomainModel 层引用
- DAL.Test: 引用 NHibernate.dll 和 Iesi.Collections.dll 程序集, nunit.framework.dll 程序集(测试框架), DomainModel 层和 DAL层引用

4. 编写 DomainModel 层

4-1. 编写持久化类

按简单传统.NET 对象(POCOs, Plain Old CLR Objects(Plain Ordinary CLR Objects))模型编程时需要持久化类,也可以说是 DTO(Data Transfer Object, 数据传送对象)模式(这是迄今为止可以工作的最简单方式)。在NHibernate 中,POCO 通过.NET 的属性机制存取数据,就可以把它映射成为数据库表。

现在为 Customer 编写持久化类来映射成为数据库表。新建一个 Customer.cs 类文件:

```
namespace DomainModel.Entities
{
    public class Customer
    {
        public virtual int CustomerId { get; set; }
        public virtual string Firstname { get; set; }
        public virtual string Lastname { get; set; }
    }
}
```

规则

- NHibernate 使用属性的 getter 和 setter 来实现持久化。
- 属性可设置为 public、internal、protected、protected internal 或 private

注意 NHibernate 默认使用代理功能,要求持久化类不是 sealed 的,而且其公共方法、属性和事件声明为 virtual。在这里,类中的字段要设置为 virtual,否则出现“failed: NHibernate.InvalidProxyTypeException : The following types may not be used as proxies: DomainModel.Entities.Customer: method get_CustomerId should be virtual, method set_CustomerId should be virtual”异常。

4-2. 编写映射文件

小提示我们要为 Microsoft Visual Studio 2008 添加编写 NHibernate 配置文件智能提示的功能。只要在下载的 NHibernate 里找到 configuration.xsd 和 nhibernate-mapping.xsd 两个文件并复制到 X:\Program Files\Microsoft Visual Studio 9.0\Xml\Schema s 目录即可。

NHibernate 要知道怎样去加载和存储持久化类的对象。这正是 NHibernate 映射文件发挥作用的地方。映射文件包含了对象/关系映射所需的元数据。元数据包含持久化类的声明和属性到数据库的映射。映射文件告诉 NHibernate 它应该访问数据库里面的哪个表及使用表里面的哪些字段。

这里，我为 Customer.cs 类编写映射文件，具体怎么编写 O/R Mapping 文件，请参考 N Hibernate 文档。新建一 XML 文件，命名为 Customer.hbm.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  assembly="DomainModel" namespace="DomainModel">
  <class name="DomainModel.Entities.Customer,DomainModel" table="Customer">
    <id name="CustomerId" column="CustomerId" type="Int32" unsaved-value="0">
      <generator class="native"></generator>
    </id>
    <property name="Firstname" column="Firstname" type="string" length="50" not-null="false"/>
    <property name="Lastname" column="Lastname" type="string" length="50" not-null="false"/>
  </class>
</hibernate-mapping>
```

注意 XML 文件的默认生成操作为“内容”，这里需要修改为“嵌入的资源”生成，使用 .NET Reflector 查看程序集:



否则出现“failed: NHibernate.MappingException : No persister for: DomainModel.Entities.Customer”异常。

5. 编写数据访问层

5-1. 辅助类

我们现在可以开始 NHibernate 了。首先，我们要从 ISessionFactory 中获取一个 ISession (NHibernate 的工作单元)。ISessionFactory 可以创建并打开新的 Session。一个 Session 代表一个单线程的单元操作。ISessionFactory 是线程安全的，很多线程可以同时访问它。ISession 不是线程安全的，它代表与数据库之间的一次操作。ISession 通过 ISessionFactory 打开，在所有的工作完成后，需要关闭。ISessionFactory 通常是个线程安全的全局对象，只需要被实例化一次。我们可以使用 GoF23 中的单例 (Singleton)

模式在程序中创建 `ISessionFactory`。这个实例我编写了一个辅助类 `SessionManager` 用于创建 `ISessionFactory` 并配置 `ISessionFactory` 和打开一个新的 `Session` 单线程的方法，之后在每个数据操作类可以使用这个辅助类创建 `ISession`。

```
public class SessionManager
{
    private ISessionFactory _sessionFactory;
    public SessionManager()
    {
        _sessionFactory = GetSessionFactory();
    }
    private ISessionFactory GetSessionFactory()
    {
        return (new Configuration()).Configure().BuildSessionFactory();
    }
    public ISession GetSession()
    {
        return _sessionFactory.OpenSession();
    }
}
```

5-2. 编写操作

在 DAL 层新建一类 `NHibernateSample.cs`，编写一方法 `GetCustomerId` 用于读取客户信息。在编写方法之前，我们需要初始化 `Session`。

```
private ISession _session;
public ISession Session
{
    set
    {
        _session = value;
    }
}
public NHibernateSample(ISession session)
{
    _session = session;
}
```


NHibernate 有不同的方法来从数据库中取回对象。最灵活的方式是使用 NHibernate 查询语言(HQL)，是完全基于面向对象的 SQL。

```
public Customer GetCustomerById(int customerId)
{
    return _session.Get<Customer>(customerId);
}
```

6. 编写数据访问层的测试

6-1. 配置 NHibernate

我们可以几种方法来保存 NHibernate 的配置，具体以后来介绍，这里我们使用 hibernate.cfg.xml 文件来配置，不过不必担心，这个文件我们可以在\src\NHibernate.Test 文件夹下找到，直接复制到 DAL.Test 中修改一下配置信息和文件输出属性就可以了。

```
<?xml version="1.0" encoding="utf-8"?>
<hibernate-configuration xmlns="urn:hibernate-configuration-2.2" >
    <session-factory>
        <property name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
        <property name="connection.connection_string">
            Data Source=.\SQLEXPRESS;Initial Catalog=NHibernateSample;
            Integrated Security=True;Pooling=False
        </property>
        <property name="adonet.batch_size">10</property>
        <property name="show_sql">>true</property>
        <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
        <property name="use_outer_join">>true</property>
        <property name="command_timeout">10</property>
        <property name="query.substitutions">>true 1, false 0, yes 'Y', no 'N'</property>
        <mapping assembly="DomainModel"/>
    </session-factory>
</hibernate-configuration>
```

注意 XML 文件的默认“复制到输出目录”为“不复制”，这里需要修改为“始终复制”。否则出现“failed: NHibernate.Cfg.HibernateConfigurationException : An exception occurred during configuration of persistence layer. ----> System.IO.FileNotFoundException : 未能找到文件“NHibernateSample\DAL.Test\bin\Debug\hibernate.cfg.xml””异常。

6-2.测试

好了，终于可以使用我们的方法了，这里新建一个测试类 NHibernateSampleFixture.cs 来编写测试用例：调用 NHibernateSample 类中 GetCustomerId 方法查询数据库中 CustomerId 为 1 的客户，判断返回客户的 Id 是否为 1。

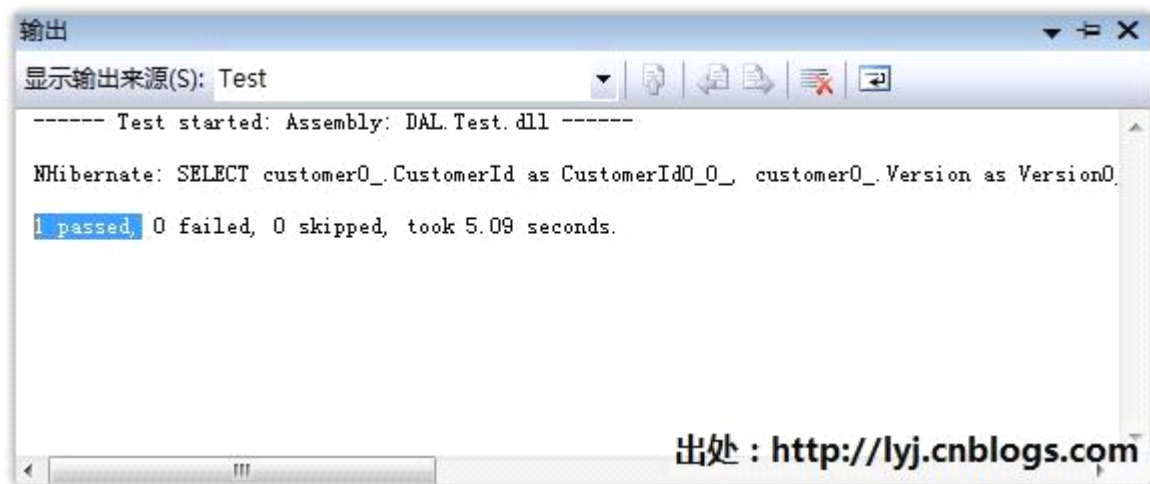
```
using NUnit.Framework;
using DomainModel.Entities;
using NHibernate;

namespace DAL.Test
{
    [TestFixture]
    public class NHibernateSampleFixture
    {
        private ISession _session;
        private SessionManager _helper;
        private NHibernateSample _sample;
        [TestFixtureSetUp]
        public void TestFixtureSetup()
        {
            _helper = new SessionManager();
        }
        [SetUp]
        public void Setup()
        {
            _session = _helper.GetSession();
            _sample = new NHibernateSample(_session);
        }
        [Test]
        public void GetCustomerByIdTest()
        {
```

```
Customer customer = _sample.GetCustomerById
(1);

int customerId = customer.CustomerId;
Assert.AreEqual(1, customerId);
}
}
}
```

我们使用 TestDriven.NET 测试一下这个方法：OK，测试通过，还输出了 SQL 语句(注：我配置 NHibernate 时设置了 show_sql=true 输出 SQL 语句)。



结语

在这篇文章中，我们使用 NHibernate 来构建了一个最基本的项目，没有体现 NHibernate 更多细节，只描绘了 NHibernate 的基本面目。当然使用 NHibernate 有各种各样的程序架构，我按照一般模式构建的。

本系列链接： [NHibernate 之旅系列文章导航](#)

NHibernate Q&A

- 欢迎加入 [NHibernate 中文社区](#)，一起讨论 NHibernate 知识！
- 请到 [NHibernate 中文社区](#) 下载本系列相关源码。

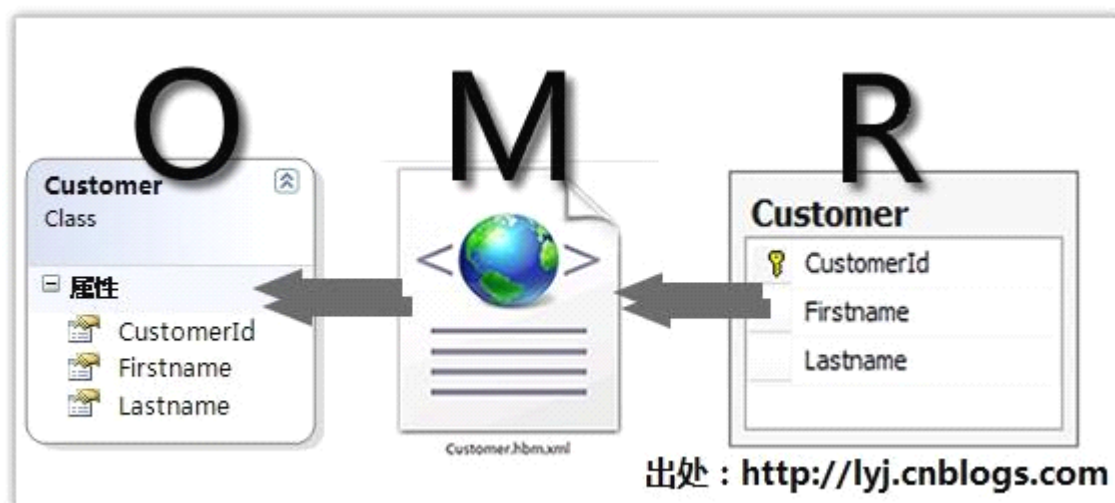
下次继续分享 NHibernate ！

NHibernate 之旅(3): 探索查询之 NHibernate 查询语言(HQL)

本节内容

- NHibernate 中的查询方法
- NHibernate 查询语言(HQL)
 - 1.from 子句
 - 2.select 子句
 - 3.where 子句
 - 4.order by 子句
 - 5.group by 子句
- 实例分析
- 结语

上一节，我们初步搭建了一个 NHibernate 程序，完成了映射 Customer 表并读取数据功能，这一节和下一节我们初步探讨一下在 NHibernate 中的查询方法。我这之前还是先回忆一下上一节完成的东西，其中一张图很多人回复说非常经典，简单明了！还是看着图。总结一下上一节三个重要的事情：建立数据库表-----编写持久化类-----编写映射文件，然后配置使用了。



NHibernate 中的查询方法

在 NHibernate 中提供了三种查询方式给我们选择：NHibernate 查询语言(HQL, NHibernate Query Language)、条件查询(Criteria API, Query By Example(QBE)是 Criteria API 的一种特殊情况)、原生 SQL(Literal SQL, T-SQL、PL/SQL)。每个人有不同的喜好和特长，可以根据自己的情况选择使用其中的一种或几种。这一节我们介绍 NHibernate 查询语言(HQL, NHibernate Query Language)。

NHibernate 查询语言(HQL)

NHibernate 查询语言(HQL, NHibernate Query Language)是 NHibernate 特有的基于面向对象的 SQL 查询语言，它具有继承、多态和关联等特性。实际上是用 OOP 中的对象和属性映射了数据库中的表和列。

例如这一句：select c.Firstname from Customer c

Customer 是数据库表，Firstname 是列；而对于 HQL：Customer 是一个对象，Firstname 是 Customer 对象的属性。相比之下 SQL 语句非常灵活，但是没有编译时语法验证支持。

本节介绍基础语法：from 子句，select 子句，where 子句，order by 子句，group by 子句并分别举出可以运行的实例。至于关联和连接，多态(polymorphism)查询，子查询在以后具体实例中学习。注意：HQL 不区分大小写。

注意：由于篇幅有限，我在这里仅仅贴出了数据访问层的代码，就是在业务逻辑层可以直接调用的方法。测试这些方法的代码就没有贴出来了，你可以下载本系列的源代码仔细看看测试这些方法的代码。这节，我们在上一节源代码的基础上，在数据访问层中新建 QueryHQL.cs 类用于编写 HQL 查询方法，在数据访问的测试层新建一 QueryHQLFixture.cs 类用于测试。

1.from 子句

顾名思义，同 SQL 语句类似：

1.简单用法：返回表中所有数据。

```
public IList<Customer> From()
{
    //返回所有 Customer 类的实例
    return _session.CreateQuery("from Customer")
        .List<Customer>();
}
```

2.使用别名：使用 `as` 来赋予表的别名，`as` 可以省略。

```
public IList<Customer> FromAlias()
{
    //返回所有 Customer 类的实例，Customer 赋予了别名 customer
    return _session.CreateQuery("from Customer as customer
")
        .List<Customer>();
}
```

3.笛卡尔积：出现多个类，或者分别使用别名，返回笛卡尔积或者称为“交叉”连接。

2.select 子句

1.简单用法：在结果集中返回指定的对象和属性。

```
public IList<int> Select()
{
    //返回所有 Customer 的 CustomerId
    return _session.CreateQuery("select c.CustomerId from Customer c")
        .List<int>();
}
```

2.数组：用 `Object[]`的数组返回多个对象和/或多个属性，或者使用特殊的 `elements` 功能，注意一般要结合 `group by` 使用。

```
public IList<object[]> SelectObject()
{
    return _session.CreateQuery("select c.Firstname, count(c.Firstname) from Customer c group by c.Firstname")
        .List<object[]>();
}
```

或者使用类型安全的.NET 对象，以后在实例中说明。

3.统计函数：用 `Object[]`的数组返回属性的统计函数的结果，注意统计函数的变量也可以是集合 `count(elements(c.CustomerId))`

```
public IList<object[]> AggregateFunction()
{
    return _session.CreateQuery("select avg(c.CustomerId),sum(c.CustomerId),count(c) from Customer c")
}
```

```
.List<object[]>();  
}
```

4. Distinct 用法: distinct 和 all 关键字的用法和语义与 SQL 相同。实例: 获取不同 Customer 的 FirstName。

```
public IList<string> Distinct()  
{  
    return _session.CreateQuery("select distinct c.Firstname f  
    rom Customer c")  
        .List<string>();  
}
```

3. where 子句

where 子句让你缩小你要返回的实例的列表范围。

```
public IList<Customer> Where()  
{  
    return _session.CreateQuery("select from Customer c whe  
    re c.Firstname='YJing'")  
        .List<Customer>();  
}
```

where 子句允许出现的表达式包括了在 SQL 中的大多数情况:

- 数学操作符: +, -, *, /
- 真假比较操作符: =, >=, <=, <>, !=, like
- 逻辑操作符: and, or, not
- 字符串连接操作符: ||
- SQL 标量函数: upper(), lower()
- 没有前缀的 (): 表示分组
- in, between, is null
- 位置参数: ?
- 命名参数: :name, :start_date, :x1

- SQL 文字: 'foo', 69, '1970-01-01 10:00:01.0'
- 枚举值或常量: Color.Tabby

4.order by 子句

按照任何返回的类或者组件的属性排序: asc 升序、desc 降序。

```
public IList<Customer> Orderby()
{
    return _session.CreateQuery("select from Customer c orde
r by c.Firstname asc,c.Lastname desc")
        .List<Customer>();
}
```

5.group by 子句

按照任何返回的类或者组件的属性进行分组。

```
public IList<object[]> Groupby()
{
    return _session.CreateQuery("select c.Firstname, count(c.
Firstname) from Customer c group by c.Firstname")
        .List<object[]>();
}
```

实例分析

好的, 以上基本的查询的确非常简单, 我们还是参考一下实例, 分析一下我们如何写 HQL 查询吧!

实例 1: 按照 FirstName 查询顾客:

```
public IList<Customer> GetCustomersByFirstname(string first
name)
{
    //写法 1
    //return _session.CreateQuery("select from Customer c wh
ere c.Firstname='" + firstname + "'")
        // .List<Customer>();
}
```



```

//写法 2:位置型参数
//return _session.CreateQuery("select from Customer c wh
ere c.Firstname=?")
//    .SetString(0, firstname)
//    .List<Customer>();

//写法 3:命名型参数(推荐)
return _session.CreateQuery("select from Customer c whe
re c.Firstname=:fn")
    .SetString("fn", firstname)
    .List<Customer>();
}

```

书写 HQL 参数有四种写法:

- 写法 1: 可能会引起 SQL 注入, 不要使用。
- 写法 2: ADO.NET 风格的?参数, NHibernate 的参数从 0 开始计数。
- 写法 3: 命名参数用:name 的形式在查询字符串中表示, 这时 IQuery 接口把实际参数绑定到命名参数。
- 写法 4: 命名的参数列表, 把一些参数添加到一个集合列表中的形式, 比如可以查询数据是否在这个集合列表中。

使用命名参数有一些好处: 命名参数不依赖于它们在查询字符串中出现的顺序; 在同一个查询中可以使用多次; 它们的可读性好。所以在书写 HQL 使用参数的时候推荐命名型参数形式。

测试一下这个方法吧: 看看数据库中 Firstname 为“YJingLee”的记录个数是否是 1 条, 并可以判断查询出来的数据的 FirstName 属性是不是“YJingLee”。

```

[Test]
public void GetCustomerByFirstnameTest()
{
    IList<Customer> customers = _queryHQL.GetCustomersB
yFirstname("YJingLee");
    Assert.AreEqual(1, customers.Count);
    foreach (var c in customers)
    {
        Assert.AreEqual("YJingLee", c.Firstname);
    }
}

```

```
}  
}
```

实例 2: 获取顾客 ID 大于 CustomerId 的顾客:

```
public IList<Customer> GetCustomersWithCustomerIdGreater  
Than(int customerId)  
{  
    return _session.CreateQuery("select from Customer c whe  
re c.CustomerId > :cid")  
        .SetInt32("cid", customerId)  
        .List<Customer>();  
}
```

结语

在这篇文章中，我们了解了 NHibernate 其中的一种查询语言 HQL，这些实例我争取写出来可以运行起来，大家下载源码看看效果，一些数据需要按个人情况修改。例如查询条件结果。下一节继续介绍另外一种查询语言！

NHibernate 之旅(4): 探索查询之条件查询(Criteria Query)

本节内容

- NHibernate 中的查询方法
- 条件查询(Criteria Query)
 - 1.创建 ICriteria 实例
 - 2.结果集限制
 - 3.结果集排序
 - 4.一些说明
- 根据示例查询(Query By Example)
- 实例分析
- 结语

上一节，我们介绍了 NHibernate 查询语言的一种：NHibernate 查询语言(HQL，NHibernate Query Language)，这一节介绍一下条件查询(Criteria API)。

NHibernate 中的查询方法

在 NHibernate 中提供了三种查询方式给我们选择：NHibernate 查询语言(HQL，NHibernate Query Language)、条件查询(Criteria API, Criteria Query)、(根据示例查询(QBE, Query By Example)是条件查询的一种特殊情况)、原生 SQL(Literal SQL, T-SQL、PL/SQL)。每个人有不同的喜好和特长，可以根据自己的情况选择使用其中的一种或几种。这一节我们介绍条件查询。

条件查询(Criteria Query)

HQL 极为强大，但是有些人希望能够动态的使用一种面向对象 API 创建查询，而不是在.NET 代码中嵌入字符串。在 NHibernate 中，提供了一种直观的、可扩展的 Criteria API。在我们键入查询语句的时候，提供了编译时的语法检查，VS 提供了强大的智能提示。如果你对 HQL 的语法感觉不是很舒服的话，用这种方法可能更容易。这种 API 也比 HQL 更可扩展。

典型用法：从 `ISession` 接口中创建 `ICriteria` 实例对象；在这个 `ICriteria` 实例对象上设置一个或多个表达式；要求 `ICriteria` 接口返回需要的列表，就是根据表达式从数据库中返回对象。

注意：由于篇幅有限，我在这里仅仅贴出了数据访问层的代码。测试这些方法的代码就没有贴出来了，你可以下载本系列的源代码仔细看看测试这些方法的代码。这些实例我争取写出来可以运行起来，大家下载源码看看效果，一些数据需要按个人数据库里的数据情况修改。例如查询条件和结果。这节，我们在上一节源代码的基础上，在数据访问层中新建 `QueryCriteriaAPI.cs` 类用于编写条件查询方法，在数据访问的测试层新建一 `QueryCriteriaAPIFixture.cs` 类用于测试。

1. 创建 `ICriteria` 实例

使用 `ISession` 接口的 `CreateCriteria` 方法创建了 `NHibernate.ICriteria` 接口一个特定的持久化类的查询实例，也可以说 `ISession` 是用来制造 `Criteria` 实例的工厂。

```
public IList<Customer> CreateCriteria()
{
    ICriteria crit = _session.CreateCriteria(typeof(Customer));
    crit.SetMaxResults(50);
    IList<Customer> customers = crit.List<Customer>();
    return customers;
}
```

例如上面的例子返回 `Customer` 对象集合，设置最大的集合数量为 50 条。

2. 结果集限制

使用 `ICriteria` 接口提供的 `Add` 方法添加 `Restrictions` 类中约束表达式可以限制一些结果集的作用。

```
public IList<Customer> Narrowing()
{
    IList<Customer> customers = _session.CreateCriteria(typeof(Customer))
        .Add(Restrictions.Like("Firstname", "YJing%"))
        .Add(Restrictions.Between("Lastname", "A%", "Y%"))
        .List<Customer>();
    return customers;
}
```

3. 结果集排序

使用 `ICriteria.Order` 对结果集排序，第二个参数 `true` 代表 `asc`，`false` 代表 `desc`。例如下面例子查询 `Customer` 对象按 `Firstname` 降序、`Lastname` 升序。

```
public IList<Customer> Order()
{
    return _session.CreateCriteria(typeof(Customer))
        .Add(Restrictions.Like("Firstname", "Y%"))
        .AddOrder(new NHibernate.Criterion.Order("Firstname", false))
        .AddOrder(new NHibernate.Criterion.Order("Lastname", true))
        .List<Customer>();
}
```

4. 一些说明

条件查询同样支持关联查询、动态关联抓取(在介绍一对多，多对多关系中阐述)，投影、聚合和分组，离线(`detached`)查询和子查询是 2.0 版新增加的内容，以后在相关知识中介绍。也可以自行参考 `NHibernate` 参考文档 13 章。

根据示例查询(Query By Example)

根据示例查询(QBE, Query By Example)是条件查询的一种特殊情况，`NHibernate.Criterion.Example` 类根据你指定的实例创造查询条件。其典型的用法：创建一个 `Example` 实例；在 `Example` 实例上设置值；根据 `Example` 和设置 `NHibernate` 返回其对象集合。

例如下面的例子，按照指定 `Customer` 查询数据库里的记录：

```
public IList<Customer> Query()
{
    Customer customerSample = new Customer() { Firstname = "YJing", Lastname = "Lee" };
    return _session.CreateCriteria(typeof(Customer))
        .Add(Example.Create(customerSample))
        .List<Customer>();
}
```

你可以自行调整 `Example` 使之更实用：

```
public IList<Customer> UseQueryByExample_GetCustomer(Customer customerSample)
```

```

{
    Example example = Example.Create(customerSample)
        .IgnoreCase()
        .EnableLike()
        .SetEscapeCharacter('&');
    return _session.CreateCriteria(typeof(Customer))
        .Add(example)
        .List<Customer>();
}

```

实例分析

实例 1: 利用 CriteriaAPI 按 Firstname 和 Lastname 查询顾客。

```

public IList<Customer> GetCustomersByFirstnameAndLastname(
    string firstname, string lastname)
{
    return _session.CreateCriteria(typeof(Customer))
        .Add(Restrictions.Eq("Firstname", firstname))
        .Add(Restrictions.Eq("Lastname", lastname))
        .List<Customer>();
}

```

测试: 调用 GetCustomersByFirstnameAndLastname 方法, 查询 Firstname 为"YJing", Lastname 为"Lee"的顾客, 判断查询结果数量是否为 1。(注: 在数据库中有符合这一个记录)

```

[Test]
public void GetCustomerByFirstnameAndLastnameTest()
{
    IList<Customer> customers =
        _queryCriteriaAPI.GetCustomersByFirstnameAndLastname(
            "YJing", "Lee");
    Assert.AreEqual(1, customers.Count);
}

```

实例 2: 利用 CriteriaAPI 获取顾客 ID 大于 CustomerId 的顾客。

```

public IList<Customer> GetCutomersWithIdGreaterThan(int customerId)

```

```
{  
    return _session.CreateCriteria(typeof(Customer))  
        .Add(Restrictions.Gt("CustomerId", customerId))  
        .List<Customer>();  
}
```

结语

好了，通过 2 篇文章的介绍，对 NHibernate 中的查询语法有了大致了解，知道了 NHibernate 中两种最主要的查询方式，还有一种原生 SQL 查询，内容不多，请参考 NHibernate 官方文档吧。多多练习！下节将介绍对对象的操作。

NHibernate 之旅(5): 探索 Insert, Update, Delete 操作

本节内容

- 操作数据概述
- 1.新建对象
- 2.删除对象
- 3.更新对象
- 4.保存更新对象
- 结语

操作数据概述

我们常常所说的一个工作单元，通常是执行 1 个或多个操作，对这些操作要么提交要么放弃/回滚。想想使用 LINQ to SQL，一切的东西都在内存中操作，只有调用了 `DataContext.SubmitChanges()` 方法才把这些改变的数据提交到数据库中，LINQ to SQL 那么提交要么回滚。

我们使用 NHibernate 也一样，如果只查询数据，不改变它的值，就不需要提交(或者回滚)到数据库。

注意：这节，我们在上一节源代码的基础上，在数据访问层中新建 `CRUD.cs` 类用于编写操作方法，在数据访问的测试层新建一 `CRUDFixture.cs` 类用于测试。

1.新建对象

简单描述：新建一个对象；调用 `ISession.Save()`；同步 `ISession`。

例子：在数据访问层编写 `CreateCustomer()` 方法，把传过来的 `Customer` 对象保存在数据库中。

```
public int CreateCustomer(Customer customer)
{
    int newid = (int)_session.Save(customer);
    _session.Flush();
    return newid;
}
```



```
}
```

我们测试这个方法，新建一个 `Customer` 对象，调用 `CreateCustomer()` 方法返回新插入的 `CustomerId`，再次根据 `CustomerId` 查询数据库是否存在这个对象。

```
[Test]
public void CreateCustomerTest()
{
    var customer = new Customer() { Firstname = "YJing", Lastname = "Lee" };
    int newIdentity = _crud.CreateCustomer(customer);
    var testCustomer = _crud.GetCustomerById(newIdentity);
    Assert.IsNotNull(testCustomer);
}
```

2. 删除对象

简单描述：获取一个对象；调用 `ISession.Delete()`；同步 `ISession`。

说明：使用 `ISession.Delete()` 会把对象的状态从数据库中移除。当然，你的应用程序可能仍然持有指向它的引用。所以，最好这样理解：`Delete()` 的用途是把一个持久化实例变成临时实例。你也可以通过传递给 `Delete()` 一个 `NHibernate` 查询字符串来一次性删除很多对象。删除对象顺序没有要求，不会引发外键约束冲突。当然，有可能引发在外键字段定义的 `NOT NULL` 约束冲突。

例子：在数据访问层编写 `DeleteCustomer()` 方法，从数据库中删除 `Customer` 对象。

```
public void DeleteCustomer(Customer customer)
{
    _session.Delete(customer);
    _session.Flush();
}
```

我们测试这个方法，在数据库中查询 `CustomerId` 为 2 的 `Customer` 对象，调用 `DeleteCustomer()` 方法删除，再次根据 `CustomerId` 查询数据库是否存在这个对象。

```
[Test]
public void DeleteCustomerTest()
{
    var customer = crud.GetCustomerById(2);
    _crud.DeleteCustomer(customer);
}
```

```
var testCustomer = _crud.GetCustomerById(2);
Assert.IsNull(testCustomer);
}
```

3.更新对象

简单描述：获取一个对象；改变它的一些属性；调用 `ISession.Update()`；同步 `ISession`。

例子：在数据访问层编写 `UpdateCustomer()`方法，修改 `Customer` 对象。

```
public void UpdateCustomer(Customer customer)
{
    _session.Update(customer);
    _session.Flush();
}
```

测试这个方法，在数据库中查询 `CustomerId` 为 1 的 `Customer` 对象并修改它的 `Firstname` 属性值，调用 `UpdateCustomer()`方法更新，再次查询数据库中 `CustomerId` 为 1 的 `Customer` 对象的 `Firstname` 值为修改之后的值。

```
[Test]
public void UpdateCustomerTest()
{
    var customer = _crud.GetCustomerById(1);
    customer.Firstname = "liyongjing";
    _crud.UpdateCustomer(customer);
    var testCustomer = _crud.GetCustomerById(1);
    Assert.AreEqual("liyongjing", customer.Firstname);
}
```

4.保存更新对象

你会不会想出这个问题？哪些是刚刚创建的对象，哪些是修改过的对象？对于刚刚创建的对象我们需要保存到数据库中，对于修改过的对象我们需要更新到数据库中。

幸好，`ISession` 可以识别出这不同的对象，并为我们提供了 `ISession.SaveOrUpdate(object)`方法

`ISession.SaveOrUpdate(object)`方法完成如下工作：

- 检查这个对象是否已经存在 `Session` 中。

- 如果对象不在，调用 `Save(object)`来保存。
- 如果对象存在，检查这个对象是否改变了。
- 如果对象改变，调用 `Update(object)`来更新。

看看下面例子说明了这种情况，在数据访问层编写 `SaveOrUpdateCustomer()`方法，保存更新 `Customer` 对象列表，依次遍历列表中的 `Customer` 对象，调用 `ISession.SaveOrUpdate(object)`方法保存更新每个 `Customer` 对象。

```
public void SaveOrUpdateCustomer(IList<Customer> customer
r)
{
    foreach (var c in customer)
    {
        _session.SaveOrUpdate(c);
    }
    _session.Flush();
}
```

测试这个方法，先在数据库中查询 `Firstname` 为 `YJing` 的 `Customer` 对象并修改它的 `Lastname` 属性值，这些对象是数据库中存在的，并改变了，然后新建 2 个 `Customer` 对象，这两个对象在数据库中不存在，是新创建的。调用 `SaveOrUpdateCustomer()`方法保存更新对象，即更新前面修改的对象和保存了后面新创建的 2 个对象。再次查询数据库中 `Firstname` 为 `YJing`，`Lastname` 为 `YongJing` 的 `Customer` 对象是否一致了。

```
[Test]
public void SaveOrUpdateCustomerTest()
{
    IList<Customer> customers = _crud.GetCustomersByFirst
name("YJing");
    foreach (var c in customers)
    {
        c.Lastname = "YongJing";
    }
    var c1 = new Customer() { Firstname = "YJing", Lastname
= "YongJing"};
    var c2 = new Customer() { Firstname = "YJing", Lastname
= "YongJing"};
    customers.Add(c1);
    customers.Add(c2);
}
```

```
int initiaIListCount = customers.Count;

_crud.SaveOrUpdateCustomer(customers);

int testListCount = _crud.GetCustomersByFirstnameAndLastname("YJing", "YongJing").Count;
Assert.AreEqual(initiaIListCount, testListCount);
}
```

结语

当然，这一节操纵对象操作，在NHibernate 中涉及了对象的状态，对象对一个特定的 **ISession** 来说，有三种状态分别是：瞬时(**transient**)对象、持久化(**persistent**)对象、游离(**detached**)对象。这一节没有说到了，以后在讨论 **Session** 的时候再介绍。

NHibernate 之旅(6): 探索 NHibernate 中的事务

本节内容

- 事务概述
- 1.新建对象
 - 【测试成功提交】
 - 【测试失败回滚】
- 2.删除对象
- 3.更新对象
- 4.保存更新对象
- 结语

上一篇我们介绍了 NHibernate 中的 Insert, Update, Delete 操作，这篇我们来看看 NHibernate 中的事务。你通过它可以提交或者回滚你的操作。

事务概述

1.NHibernate 中的事务(Transactions)

简单描述：要求 ISession 使用事务；做一些操作；提交或者回滚事务。

写成代码就像这样：

```
ITransaction tx = _session.BeginTransaction();  
//一些保存、更新、删除等操作  
tx.Commit();
```

实际上在 NHibernate 使用事务要使用 using 强制资源清理和异常机制，一般像这样：

```
using (ITransaction tx = _session.BeginTransaction())  
{  
    try  
    {  
        //一些保存、更新、删除等操作
```

```
        tx.Commit();
    }
    catch (HibernateException)
    {
        tx.Rollback();
        throw;
    }
}
```

2.什么时候使用事务?

回答是：在任何时候都要使用事务，即使是在读取、查询数据的时候，为什么呢？因为你不清楚数据库什么时候操作失败，如何恢复原来数据。而 NHibernate 中的事务（可以通过 tx.Rollback() 方法），帮助我们完成这些事情。

下面看看例子，我们修改上篇的 Insert、Update、Delete 操作：

1.新建对象

```
public int CreateCustomerTransaction(Customer customer)
{
    using (ITransaction tx = _session.BeginTransaction())
    {
        try
        {
            int newId = (int)_session.Save(customer);
            _session.Flush();
            tx.Commit();
            return newId;
        }
        catch (HibernateException)
        {
            tx.Rollback();
            throw;
        }
    }
}
```

这篇以新建对象为例，分别从成功提交和失败回滚两个角度来测试这个方法。

【测试成功提交】

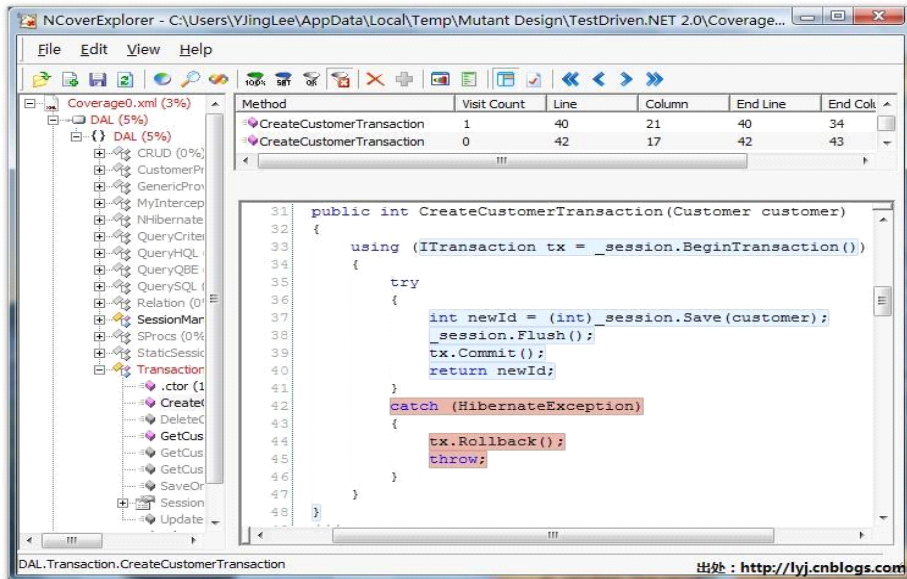
首先写一个测试用例，假设这个测试可以运行成功：

```
[Test]
public void CreateCustomerTransactionTest()
{
    var customer = new Customer() { Firstname = "YJing", Lastname = "Lee" };
    int newIdentity = _transaction.CreateCustomerTransaction(customer);
    var testCustomer = _transaction.GetCustomerById(newIdentity);
    Assert.IsNotNull(testCustomer);
}
```

测试这个方法，使用 TestDriven.NET 集成的 NCover(分析代码的覆盖率)查看代码运行覆盖率，在这个测试方法上右击选择“Test With”—“Coverage”，如下图所示：



这时自动打开 NCoverExplorer(查看代码覆盖率的分析结果)，我们可以看到 CreateCustomerTransaction 方法运行覆盖情况，我们发现这个方法通过事务成功提交了操作并返回新的 Id。分析结果效果图如下所示：



【测试失败回滚】

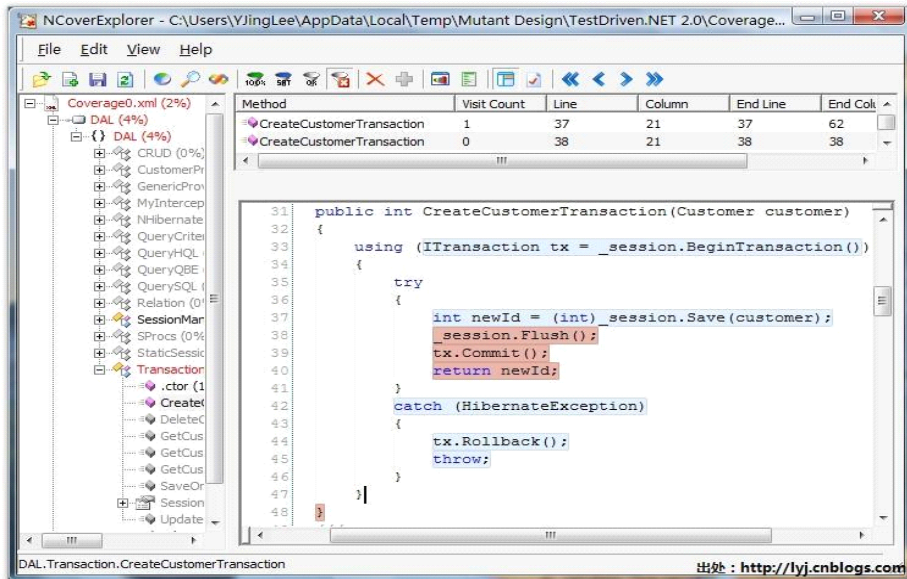
我们在写一个失败回滚的测试，由于我认为设置了一个“将截断字符串或二进制数据”错误，这时必须在测试方法上指定测试预期的异常。

```

[Test]
[ExpectedException(typeof(NHibernate.HibernateException))]
public void CreateCustomerThrowExceptionOnFailTest()
{
    var customer = new Customer()
    {
        Firstname = "01234567890123456789012345678901
2345678901234567890123456789",
        Lastname = "YJingLee"
    };
    _transaction.CreateCustomerTransaction(customer);
}

```

同理按上面的步骤测试这个方法看看 CreateCustomerTransaction 方法运行情况，由于出现错误(这里是“将截断字符串或二进制数据”错误)，所以系统抛出了 HibernateException 异常，此时系统发生回滚。分析结果效果图如下所示：



2. 删除对象

我们修改上例中的删除对象的代码，如下所示：

```
public void DeleteCustomerTransaction(Customer customer)
{
    using (ITransaction tx = _session.BeginTransaction())
    {
        try
        {
            session.Delete(customer);
            session.Flush();
            tx.Commit();
        }
        catch (HibernateException)
        {
            tx.Rollback();
            throw;
        }
    }
}
```

3. 更新对象

我们修改上例中的更新对象的代码，如下所示：

```
public void UpdateCustomerTransaction(Customer customer)
{
    using (ITransaction tx = _session.BeginTransaction())
    {
        try
        {
            _session.Update(customer);
            _session.Flush();
            tx.Commit();
        }
        catch (HibernateException)
        {
            tx.Rollback();
            throw;
        }
    }
}
```

4. 保存更新对象

我们修改上例中的保存更新对象的代码，如下所示：

```
public void SaveOrUpdateCustomersTransaction(IList<Customer> customers)
{
    using (ITransaction tx = _session.BeginTransaction())
    {
        try
        {
            foreach (Customer c in customers)
                _session.SaveOrUpdate(c);
            _session.Flush();
            tx.Commit();
        }
        catch (HibernateException)
        {
            tx.Rollback();
        }
    }
}
```

```
        throw;  
    }  
}  
}
```

好了，由于篇幅有限，上面三个方法在这里我就不测试了，大家可以参考创建对象测试的步骤来测试一下其他几个方法吧！

结语

感觉这节内容很少的样子，在 **NHibernate** 官方文档中对事务讲解的并不多，自己挖空心思也就挤了这么多东西。不过在这一节带领大家学会了测试工具 **TestDriven.NET** 的另一个功能就是怎么查看代码运行覆盖率，还是有一点收获的哦。下一节想继续深入事务话题一起讨论 **NHibernate** 中的并发控制，到现在还没有想好怎么写呢，希望大家对这个系列给出意见和建议。谢谢支持！

NHibernate 之旅(7): 初探 NHibernate 中的并发控制

本节内容

- 什么是并发控制?
 - 悲观并发控制(Pessimistic Concurrency)
 - 乐观并发控制(Optimistic Concurrency)
- NHibernate 支持乐观并发控制
- 实例分析
- 结语

什么是并发控制?

当许多人试图同时修改数据库中的数据时，必须实现一个控制系统，使一个人所做的修改不会对他人所做的修改产生负面影响。这称为并发控制。

简单的理解就是 2 个或多个用者同时编辑相同的数据。这里的用者可能是：实际用户、不同服务、不同的代码段（使用多线程），及其在断开式和连接式情况下可能发生的情况。

并发控制理论根据建立并发控制的方法而分为两类：

悲观并发控制(Pessimistic Concurrency)

一个锁定系统，可以阻止用户以影响其他用户的方式修改数据。如果用户执行的操作导致应用了某个锁，只有这个锁的所有者释放该锁，其他用户才能执行与该锁冲突的操作。这种方法之所以称为悲观并发控制，是因为它主要用于数据争用激烈的环境中，以及发生并发冲突时用锁保护数据的成本低于回滚事务的成本的环境中。

简单的理解通常通过“独占锁”的方法。获取锁来阻塞对于别的进程正在使用的数据的访问。换句话说，读者和写者之间是会互相阻塞的，这可能导致数据同步冲突。

乐观并发控制(Optimistic Concurrency)

在乐观并发控制中，用户读取数据时不锁定数据。当一个用户更新数据时，系统将进行检查，查看该用户读取数据后其他用户是否又更改了该数据。如果其他用户更新了数据，将产生一个错误。一般情况下，收到错误信息的用户将回滚事务并重新开始。这种方法之所以称为乐

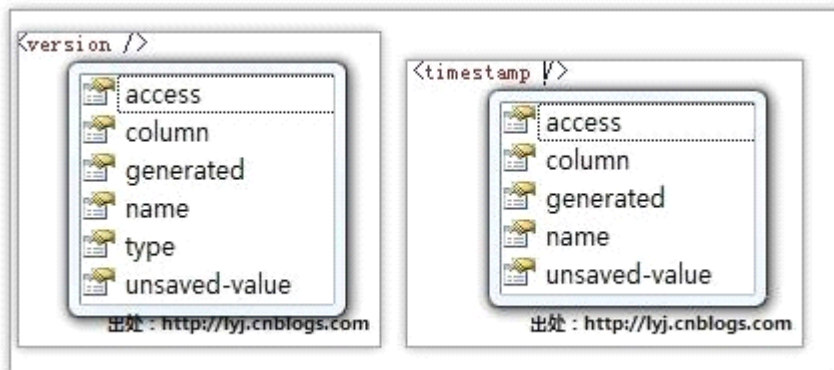
观并发控制，是由于它主要在以下环境中使用：数据争用不大且偶尔回滚事务的成本低于读取数据时锁定数据的成本。

(以上摘自 SQL Server2008 MSDN 文档)

NHibernate 支持乐观并发控制

NHibernate 提供了一些方法来支持乐观并发控制：在映射文件中定义了 `<version>` 节点和 `<timestamp>` 节点。其中 `<version>` 节点用于版本控制，表明表中包含附带版本信息的数据。`<timestamp>` 节点用于时间戳跟踪，表明表中包含时间戳数据。时间戳本质上是一种对乐观锁定不是特别安全的实现。但是通常而言，版本控制方式是首选的方法。当然，有时候应用程序可能在其他方面使用时间戳。

下面用两幅图显示这两个节点映射属性：



看看它们的意义：

- `access`(默认为 `property`): NHibernate 用于访问特性值的策略。
- `column`(默认为特性名): 指定持有版本号的字段名或者持有时间戳的字段名。
- `generated`: 生成属性，可选 `never` 和 `always` 两个属性。
- `name`: 持久化类的特性名或者指定类型为 .NET 类型 `DateTime` 的特性名。
- `type`(默认为 `Int32`): 版本号的类型，可选类型为 `Int64`、`Int32`、`Int16`、`Ticks`、`Timestamp`、`TimeSpan`。注意: `<timestamp>` 和 `<version type="timestamp">` 是等价的。
- `unsaved-value`(在版本控制中默认是“敏感”值，在时间戳默认是 `null`): 表示某个实例刚刚被实例化(尚未保存)时的版本特性值，依靠这个值就可以把这种情况和已经在先前的会话中保存或装载的游离实例区分开来。(undefined 指明使用标识特性值进行判断)

实例分析

下面用一个例子来实现乐观并发控制，这里使用 Version 版本控制。

1.修改持久化 Customer 类：添加 Version 属性

```
public class Customer
{
    public virtual int CustomerId { get; set; }
    //版本控制
    public virtual int Version { get; set; }
    public virtual string Firstname { get; set; }
    public virtual string Lastname { get; set; }
}
```

2.修改映射文件：添加 Version 映射节点

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="DomainModel" namespace="DomainModel">
    <class name="DomainModel.Entities.Customer,DomainModel" table="Customer">
        <id name="CustomerId" column="CustomerId" type="Int32" unsaved-value="0">
            <generator class="native"></generator>
        </id>
        <version name="Version" column="Version" type="integer" unsaved-value="0"/>
        <property name="Firstname" column="Firstname" type="string" length="50" not-null="false"/>
        <property name="Lastname" column="Lastname" type="string" length="50" not-null="false"/>
    </class>
</hibernate-mapping>
```

3.修改数据库，添加 Version 字段

具体参数: [Version] [int] NOT NULL 默认值为 1,当然了修改数据库是最原始的方式了,如果你会使用 SchemaExport, 可以直接利用持久化类和映射文件生成数据库, 以后在介绍如何使用这个。

4.并发更新测试

在测试之前,我们先看看数据库中什么数据, 预知一下:



	CustomerId	Version	Firstname	Lastname
1	1	1	YJing	Lee
2	2	1	li	yongjing

出处: <http://lyj.cnblogs.com>

编写并发更新测试代码:

查询 2 次 CustomerId 为 1 的客户, 这里就是上面的第一条数据, 第一个修改为"CnBlog s", 第二个修改为"www.cnblogs.com", 两者同时更新提交。你想想发生什么情况?

```
[Test]
public void UpdateConcurrencyViolationCanotThrowException()
{
    Customer c1 = _transaction.GetCustomerById(1);
    Customer c2 = _transaction.GetCustomerById(1);
    c1.Name.Firstname = "CnBlogs";
    c2.Name.Firstname = "www.cnblogs.com";

    _transaction.UpdateCustomerTransaction(c1);
    _transaction.UpdateCustomerTransaction(c2);
}
```

让我们去看看数据库吧, 一目了然:



	CustomerId	Version	Firstname	Lastname
1	1	2	www.cnblogs.com	Lee
2	2	1	li	yongjing

出处: <http://lyj.cnblogs.com>

我们发现 CustomerId 为 1 的客户更新了 FirstName 数据，并且 Version 更新为 2。你知道什么原理了吗？看看这步 NHibernate 生成的 SQL 语句(我的可能比你的不一样)：先查询数据库，在直接更新数据，看看 NHibernate 多么实在，明显做了一些优化工作。

```
SELECT customer0_.CustomerId as CustomerId3_0_,
customer0_.Version as Version3_0_,
customer0_.Firstname as Firstname3_0_,
customer0_.Lastname as Lastname3_0_,
customer0_1_.OrderDiscountRate as OrderDis2_4_0_,
customer0_1_.CustomerSince as Customer3_4_0_,
case
  when customer0_1_.CustomerId is not null then 1
  when customer0_.CustomerId is not null then 0
end
as clazz_0_ FROM Customer
customer0_ left outer join PreferredCustomer customer0_1_
on customer0_.CustomerId=customer0_1_.CustomerId
WHERE customer0_.CustomerId=@p0; @p0 = '1'

UPDATE Customer SET Version = @p0, Firstname = @p1, Last
name = @p2
WHERE CustomerId = @p3 AND Version = @p4;
@p0 = '2', @p1 = 'www.cnblogs.com', @p2 = 'Lee', @p3 = '1',
@p4 = '1'
```

5.并发删除测试

我们再来编写一个测试用于并发删除。查询 2 次 CustomerId 为 2 的客户，这里就是上面的第二条数据，两者同时删除数据。你想想发生什么情况？

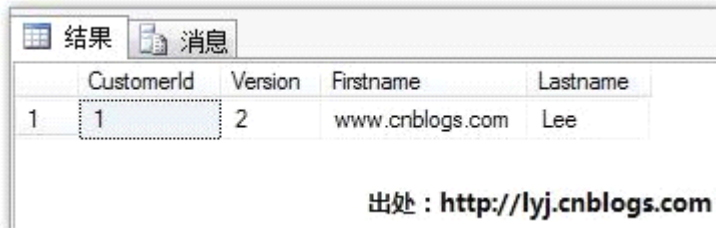
```
[Test]
[ExpectedException(typeof(NHibernate.StaleObjectStateException))]
public void DeleteConcurrencyViolationCannotThrowException()
{
    Customer c1 = _transaction.GetCustomerById(2);
    Customer c2 = _transaction.GetCustomerById(2);

    _transaction.DeleteCustomerTransaction(c1);
```



```
_transaction.DeleteCustomerTransaction(c2);  
}
```

同理，看看数据库里的数据，第二条数据不见了。



CustomerId	Version	Firstname	Lastname
1	2	www.cnblogs.com	Lee

出处：<http://lyj.cnblogs.com>

其生成 SQL 的查询语句同上面一样，只是一条删除语句：

```
DELETE FROM Customer WHERE CustomerId = @p0 AND Version = @p1; @p0 = '2', @p1 = '1'
```

好了，这里通过两个简单的实例说明了在 NHibernate 中对并发控制的支持。相信有了一定的了解，大家也可以编写一些有趣的测试来试试 NHibernate 中的乐观并发控制。

结语

这一篇我们初步探索了 NHibernate 中的并发控制，并用一个典型的实例分析了具体怎么做。我想这只是蜻蜓点水，更多的乐趣就自己探索吧。比如在不同的 Session 中的并发啊，更新啊，删除啊.....

NHibernate 之旅(8): 巧用组件之依赖对象

本节内容

- 引入
- 方案 1: 直接添加
- 方案 2: 巧用组件
- 实例分析
- 结语

引入

通过前面 7 篇的学习, 有点乏味了~~~这篇来学习一个技巧, 大家一起想想如果我要在 Customer 类中实现一个 Fullname 属性(就是 Firstname 和 Lastname 的组合)该怎么做呢?

方案 1: 直接添加

“我知道! 修改 Customer 类, 添加一个 Fullname 属性! 即 Customer.Fullname!”

“恩, 完全正确.....”

“这就意味着在 Customer 类中把 Firstname 和 Lastname 两个属性重新修改组合为 Full name 属性。这样的话, 如果有其它的类(像 Vendor、Shiper)使用了 Firstname 和 Last name 两个属性, 这就需要修改很多业务逻辑。那你的麻烦可就大了, 还有什么方法吗?”

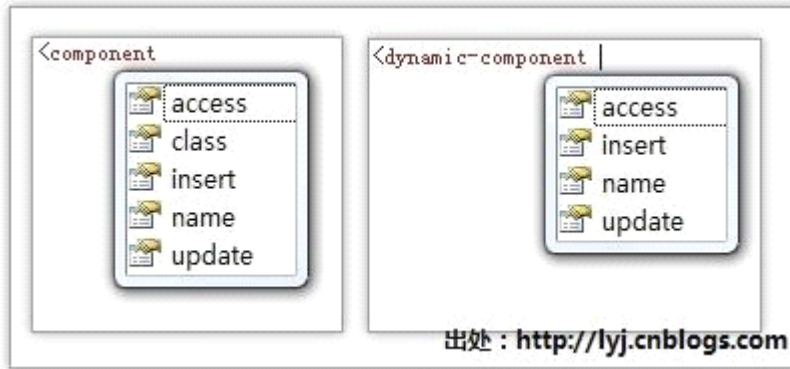
“.....”

方案 2: 巧用组件

NHibernate 中, 提供了组件(Component)和动态组件来帮助完成这件事情。其实组件在 NHibernate 中为了不同目的被重复使用。这里我们使用它来**依赖对象**。

映射文件中, <component>元素把子对象的一些属性映射为父类对应的表的一些字段。然后, 组件可以定义它们自己的属性、组件或者集合。

下面用两幅图显示组件和动态组件两个节点映射属性:



看看这些映射属性：

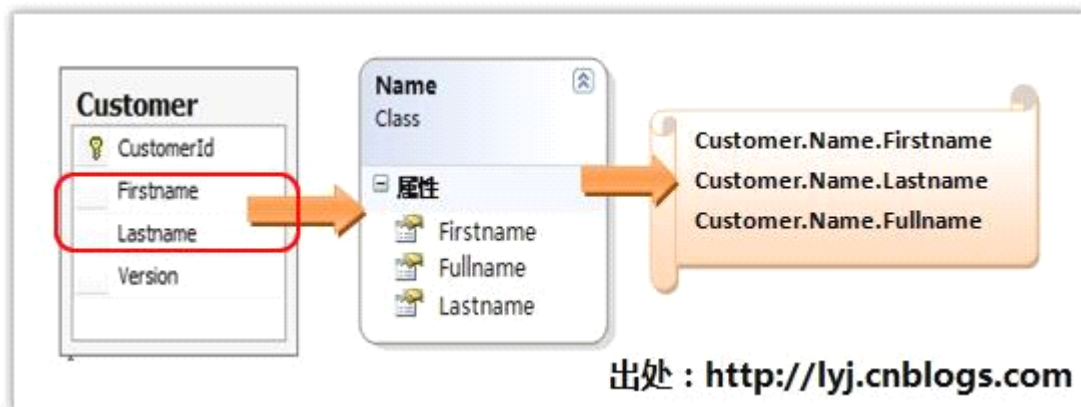
- **access**(默认 property): NHibernate 用来访问属性的策略
- **class**(默认通过反射得到的属性类型): 组件(子)类的名字
- **insert**: 被映射的字段是否出现在 SQL 的 INSERT 语句中
- **name**: 属性名 propertyName
- **update**: 被映射的字段是否出现在 SQL 的 UPDATE 语句中
- **<property>**子元素: 为组件(子)类的一些属性与表字段之间建立映射
- **<parent>**子元素: 在组件类内部就可以有一个指向其容器的实体的反向引用

<dynamic-component> 元素允许一个 IDictionary 作为组件映射，其中属性名对应字典中的键。这又是使用组件的另一种用法。

知道上面的知识，我们该想想上面的问题该如何利用组件来实现了吧。

实例分析

我们用一幅图来展示我们这节所说的一切：



开始动手吧！

1.新建 Name 类

```
namespace DomainModel.Entities
{
    public class Name
    {
        public string Firstname { get; set; }
        public string Lastname { get; set; }
        public string Fullname
        {
            get
            {
                return Firstname + " " + Lastname;
            }
        }
    }
}
```

简单的说，这个类用于组合 Fullname 属性。

2.修改 Customer 类

```
namespace DomainModel.Entities
{
    public class Customer
    {
        public virtual int CustomerId { get; set; }
        public virtual int Version { get; set; }
    }
}
```

```

    public virtual Name Name { get; set; }
}
}

```

修改 Customer 类，去除原来的 Firstname 和 Lastname 属性，添加 Name 属性。这时 Name 作为 Customer 的一个组成部分。需要注意的是：和原来 Firstname 和 Lastname 属性一样，需要对 Name 的持久化属性定义 getter 和 setter 方法，但不需要实现任何的接口或声明标识符字段。

3.修改 Customer 映射

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                    assembly="DomainModel" namespace="DomainModel">
  <class name="DomainModel.Entities.Customer,DomainModel" table="Customer">
    <id name="CustomerId" column="CustomerId" type="Int32" unsaved-value="0">
      <generator class="native"></generator>
    </id>
    <version name="Version" column="Version" type="integer" unsaved-value="0"/>
    <component name="Name" class="DomainModel.Entities.Name,DomainModel">
      <property name="Firstname" column="Firstname" type="string"
                length="50" not-null="false"
                unique-key="UC_CustomerName"/>
      <property name="Lastname" column="Lastname" type="string"
                length="50" not-null="false"
                unique-key="UC_CustomerName"/>
    </component>
  </class>
</hibernate-mapping>

```

首先定义 Component 的一些属性，指定属性名和组件映射的类名。再使用 <property> 子元素，为 Name 类的 Firstname、Lastname 属性与表字段之间建立映射。是不是很简单~~

这时 Customer 表中还是 CustomerId、Version、Firstname、Lastname 字段。完全不需要修改数据库表结构哦。

这里需要注意两点：

1. 就像所有的值类型一样，组件不支持共享引用。组件的值为空从语义学上来讲是专有的。每当重新加载一个包含组件的对象，如果组件的所有字段为空，那么 NHibernate 将假定整个组件为空。对于绝大多数目的，这样假定是没有问题的。
2. 组件的属性可以是 NHibernate 类型(包括集合、多对一关联以及其它组件)。嵌套组件不应该作为特殊的应用被考虑。NHibernate 趋向于支持设计细粒度的对象模型。

4.编写方法

这时，我们需要修改或者重新编写新的方法来实现我们想要的逻辑。

```
public IList<Customer> ReturnFullName(string firstname, string lastname)
{
    return _session
        .CreateQuery("select from Customer c where c.Name.Firstname=:fn and c.Name.Lastname=:ln")
        .SetString("fn", firstname)
        .SetString("ln", lastname)
        .List<Customer>();
}
```

现在，我们访问 Customer 的 Firstname、Lastname 属性，只需要在原来的基础上通过 Name 访问，例如上面修改的情况，看看上面图片上怎么访问的吧，一目了然。

如果我们要添加一个 Customer 怎么办呢？代码片段如下所示：

```
var customer = new Customer() { Name = new Name() { Firstname = "YJing", Lastname = "Lee" } };
```

5.测试方法

有了上面的方法，我们编写一个测试用例测试一下这个方法吧：看看结果测试成功，OK。

```
[Test]
public void ReturnFullNameTest()
{
```

```
IList<Customer> customers = _relation.ReturnFullName("
YJing", "Lee");
foreach (Customer c in customers)
{
    Assert.AreEqual("YJing Lee", c.Name.Fullname);
}
}
```

结语

这一篇向大家介绍一个使用组件技巧，通过组件可以改善我们的对象模型，而数据库结构不需要变化。通过这一篇的技巧，利用组件来映射来依赖对象，可以非常连贯的引入 **NHibernate** 中的多表映射关系、集合等内容，这些才是 **NHibernate** 中的亮点，就连 **LINQ** 都比不过它。从下篇开始就来学习 **NHibernate** 中的闪光点。

NHibernate 之旅(9): 探索父子关系(一对多关系)

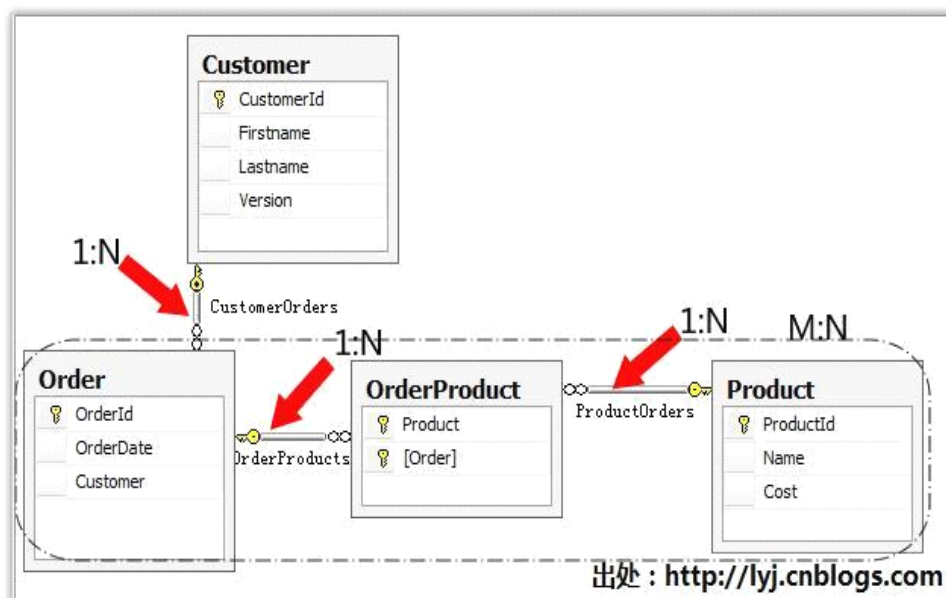
本节内容

- 引入
- NHibernate 中的集合类型
- 建立父子关系
- 父子关联映射
- 结语

引入

通过前几篇文章的介绍，基本上了解了 NHibernate，但是在 NHibernate 中映射关系是 NHibernate 中的亮点，也是最难掌握的技术。从这篇开始学习这些东西，我将图文结合来说明这里奥秘的知识。

前几篇，我们的例子只使用了一个简单的 Customer 对象。但是在客户/订单/产品的经典组合中，他们的关系非常复杂？让我们先回顾在第二篇中建立的数据模型。



在图上，我已经清晰的标注了表之间的关系，首先分析 Customer 和 Order 之间的“外键关系”或者称作“父子关系”、“一对多关系”。在分析之前先初步了解 NHibernate 中的集合。

NHibernate 中的集合类型

NHibernate 支持/定义的几种类型的集合:

Bag: 对象集合, 每个元素可以重复。例如{1,2,2,6,0,0}, 在.Net 中相当于 IList 或者 IList<T>实现。

Set: 对象集合, 每个元素必须唯一。例如{1,2,5,6}, 在.Net 中相当于 ISet 或者 ISet<T>实现, Iesi.Collections.dll 程序集提供 ISet 集合。

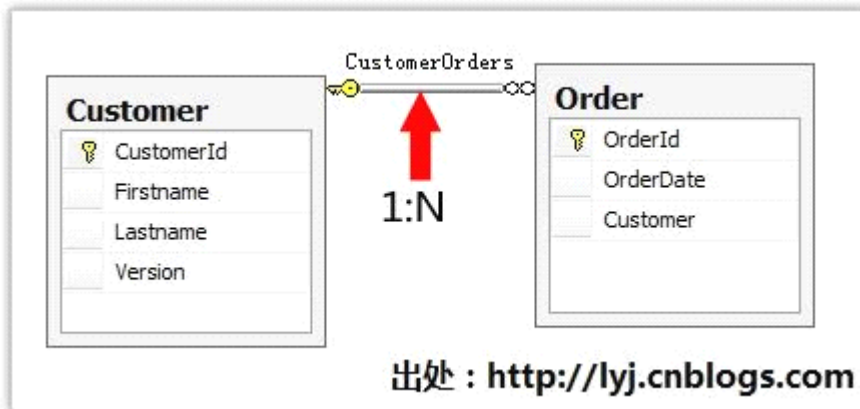
List: 整数索引对象集合, 每个元素可以重复。例如{{1,"YJingLee"},{2,"CnBlogs"},{3,"LiYongJing"}}, 在.Net 中相当于 ArrayList 或者 List<T>实现。

Map: 键值对集合。例如{{"YJingLee",5},{ "CnBlogs",7},{ "LiYongJing",6}}, 在.Net 中相当于 Hashtable 或者 IDictionary<Tkey,TValue> 实现。

实际上, 我们大多数情况下使用 Set 集合类型, 因为这个类型和关系型数据库模型比较接近。

建立父子关系

直接看下面一幅图的两张表:



上面两张表关系表达的意思是: Customer 有一个或多个 Orders, Orders 属于一个 Customer。一般而言, 我们称 Customer 为“父”, Order 称为“子”。Customer 和 Order 之间关系就有几种说法: “外键关系”、“父子关系”、“一对多关系”都可以。

1.Customer 有一个或多个 Orders

在对象模型中: 在 Customer 类中把 Orders 作为一个集合, 这时可以说 Customer 对象包含了 Orders 集合。在.NET 中通常这样表述:

```
public class Customer
```

```

{
    //.....
    public IList<Order> Orders{ get; set; }
}

```

访问对象方式：通过子集合访问：Customer.Orders[...]

在 NHibernate 中，通常而言使用 Iesi.Collections.dll 程序集中的 ISet 集合，现在修改 Customer.cs 类，首先需要引用这个程序集，Customer.cs 类代码如下：

```

using Iesi.Collections.Generic;

namespace DomainModel.Entities
{
    public class Customer
    {
        public virtual int CustomerId { get; set; }
        public virtual string Firstname { get; set; }
        public virtual string Lastname { get; set; }
        //一对多关系：Customer 有一个或多个 Orders
        public virtual ISet<Order> Orders { get; set; }
    }
}

```

2.Order 属于一个 Customer

在对象模型中：在 Order 类中把 Customer 作为单一对象，这时可以说 Order 对象包含了一个 Customer。在 .NET 中通常这样表述：

```

public class Order
{
    //.....
    public Customer Customer{ get; set; }
}

```

其访问对象方式：通过父对象成员访问：Order.Customer

我们在项目 DomainModel 层的 Entities 文件夹中新建 Order.cs 类，编写代码如下：

```

namespace DomainModel.Entities
{
    public class Order

```

```

    {
        public virtual int OrderId { get; set; }
        public virtual DateTime OrderDate { get; set; }
        //多对一关系: Orders 属于一个 Customer
        public virtual Customer Customer { get; set; }
    }
}

```

好了，我们现在完成持久类了，下面看看这两个类如何映射。

父子关联映射

在 NHibernate 中，我们可以通过映射文件来关联对象之间的关系。映射文件定义了：

- 对象之间关系：一对一、一对多、多对一、多对多关系。
- 在关系中控制级联行为(Cascade behavior)：级联更新、级联删除
- 父子之间的双向导航(bidirectional navigation)

1.父实体映射

父实体(Customer)映射定义了：

- 集合类型(Bag、Set、List、Map)
- 在保存、更新、删除操作时的级联行为
- 关联的控制方向：
 - Inverse="false"(默认)：父实体负责维护关联关系
 - Inverse="true"：子实体负责维护关联关系
- 与子实体关联的关系(一对多、多对一、多对多)

这些具体的设置是 NHibernate 中的难点所在，以后慢慢讨论这些不同设置下的奥秘之处。

这一篇初步建立 Customer 与 Order 的一对多关系，修改 Customer.hbm.xml 映射文件如下：

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"

```

```

        assembly="DomainModel" namespace="Do
mainModel">

    <class name="DomainModel.Entities.Customer,DomainMod
el"
        table="Customer">
        <id name="CustomerId" column="CustomerId" type="Int
32"
            unsaved-value="0">
            <generator class="native"></generator>
            </id>
            <property name="Firstname" column="Firstname" type=
"string"
                length="50" not-null="false"/>
            <property name="Lastname" column="Lastname" type="
string"
                length="50" not-null="false"/>
            <!--一对多关系: Customer 有一个或多个 Orders-->
            <set name="Orders" table="`Order`" generic="tru
e" inverse="true">
                <key column="Customer" foreign-key="FK_Cu
stomerOrders"/>
                <one-to-many class="DomainModel.Entities.Or
der,DomainModel"/>
            </set>
        </class>
    </hibernate-mapping>

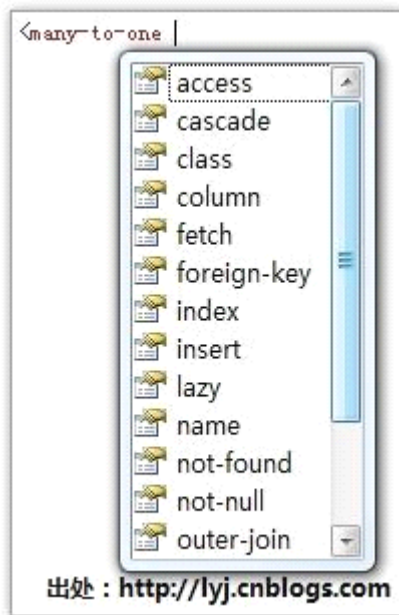
```

可以看到，在“父”端映射使用 Set 元素，标明属性名称、表名、子实体负责维护关联关系。

2.子实体映射

子实体(Order)映射定义的东西就是父实体少了：与父实体关联的(多对一、一对多、多对多) 关系，并用一个指针来导航到父实体。

在“子”端通过 many-to-one 元素定义与“父”端的关联，从“子”端角度看这种关系模型是多对一关联(实际上是对 Customer 对象的引用)。下面看看 many-to-one 元素映射属性：



看看这些映射属性具体有什么意义:

- **access**(默认 `property`): 可选 `field`、`property`、`nosetter`、`ClassName` 值。NHibernate 访问属性的策略。
- **cascade**(可选): 指明哪些操作会从父对象级联到关联的对象。可选 `all`、`save-update`、`delete`、`none` 值。除 `none` 之外其它将使指定的操作延伸到关联的(子)对象。
- **class**(默认通过反射得到属性类型): 关联类的名字。
- **column**(默认属性名): 列名。
- **fetch**(默认 `select`): 可选 `select` 和 `join` 值, `select`: 用单独的查询抓取关联; `join`: 总是用外连接抓取关联。
- **foreign-key**: 外键名称, 使用 `SchemaExport` 工具生成的名称。
- **index**:
- **update, insert**(默认 `true`): 指定对应的字段是否包含在用于 `UPDATE` 或 `INSERT` 的 SQL 语句中。如果二者都是 `false`, 则这是一个纯粹的“外源性(`derived`)”关联, 它的值是通过映射到同一个(或多个)字段的某些其他特性得到或者通过触发器其他程序得到。
- **lazy**: 可选 `false` 和 `proxy` 值。是否延迟, 不延迟还是使用代理延迟。

- name: 属性名称 propertyName。
- not-found: 可选 ignore 和 exception 值。找不到忽略或者抛出异常。
- not-null: 可选 true 和 false 值。
- outer-join: 可选 auto、true、false 值。
- property-ref(可选): 指定关联类的一个属性名称, 这个属性会和外键相对应。如果没有指定, 会使用对方关联类的主键。这个属性通常在遗留的数据库系统使用, 可能有外键指向对方关联表的某个非主键字段 (但是应该是一个唯一关键字) 的情况下, 是非常不好的关系模型。比如说, 假设 Customer 类有唯一的 CustomerId, 它并不是主键。这一点在 NHibernate 源码中有了充分的体验。
- unique: 可选 true 和 false 值。控制 NHibernate 通过 SchemaExport 工具生成 DDL 的过程。
- unique-key(可选): 使用 DDL 为外键字段生成一个唯一约束。

我们来建立“子”端到“父”端的映射, 新建 Order.hbm.xml 文件, 编写代码如下:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:hibernate-mapping-2.2"
                    assembly="DomainModel" namespace="DomainModel">

    <class name="DomainModel.Entities.Order,DomainModel" table="`Order`" >
        <id name="OrderId" column="OrderId" type="Int32" unsaved-value="0">
            <generator class="native" />
        </id>
        <property name="OrderDate" column="OrderDate" type="DateTime"
                not-null="true" />
        <!--多对一关系: Orders 属于一个 Customer-->
        <many-to-one name="Customer" column="CustomerId" not-null="true"
                    class="DomainModel.Entities.Customer,
                    DomainModel"
                    foreign-key="FK_CustomerOrders" />
    </class>
</hibernate-mapping>
```

```
</class>  
</hibernate-mapping>
```

关于如何关联看看上面的属性就一目了然了。

结语

这一篇建立 **Customer** 和 **Order** 之间一对多关联关系，并初步了解 **NHibernate** 中的集合。下一篇继续以这一篇为基础，一步一步去挖掘，这是我一贯写法，介绍在 **NHibernate** 中使用原生 **SQL**、**HQL**、**Criteria API** 三种查询方式来关联查询。当然了，从这篇开始，研究的东西就更多了，像查询、加载机制、代理机制、再议并发控制、**NHibernate** 提供的有用类等等。

NHibernate 之旅(10): 探索父子(一对多)关联查询

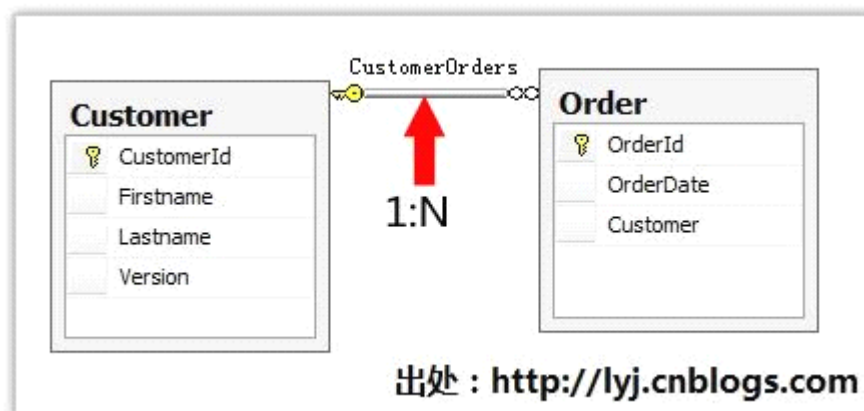
本节内容

- 关联查询引入
- 一对多关联查询
 - 1.原生 SQL 关联查询
 - 2.HQL 关联查询
 - 3.Criteria API 关联查询
- 结语

关联查询引入

在 NHibernate 中提供了三种查询方式给我们选择: NHibernate 查询语言(HQL, NHibernate Query Language)、条件查询(Criteria API, Query By Example(QBE)是 Criteria API 的一种特殊情况)、原生 SQL(Literal SQL, T-SQL、PL/SQL)。这一节分别使用这三种方式来关联查询。

首先看看上一篇我们为 Customer 和 Order 建立的父子关系:



一对多关联查询

1.原生 SQL 关联查询

在关系模型中: 可以使用子表作为内连接查询 Customer, 像这样:


```
select * from Customer c inner join Order o on c.CustomerId=o.CustomerId where o.CustomerId=<id of the Customer>
```

使用父表作为内连接查询 Order，像这样：

```
select * from Oder o inner join Customer c on o.CustomerId=c.CustomerId where o.OrderId=<id of the Order>
```

下面我们来看看在 NHibernate 中使用原生 SQL 查询。这篇来完成查询订单在 orderData 之后的顾客列表不同查询的写法。

```
public IList<Customer> UseSQL_GetCustomersWithOrders(DateTime orderDate)
{
    return _session.CreateSQLQuery("select distinct {customer.*} from Customer {customer}" +
        " inner join [Order] o on o.Customer={customer}.CustomerId where o.OrderDate > :orderDate")
        .AddEntity("customer", typeof(Customer))
        .SetDateTime("orderDate", orderDate)
        .List<Customer>();
}
```

具体情况是：实例化 IQuery 接口；使用 ISession.CreateSQLQuery()方法，传递的参数是 SQL 查询语句；{Customer.*}标记是 Customer 所有属性的简写。使用 AddEntity 查询返回的持久化类，SetDateTime 设置参数，根据不同类型，方法名不同。

2.HQL 关联查询

查询订单在 orderData 之后的顾客列表的 HQL 关联查询写法：

```
public IList<Customer> UseHQL_GetCustomersWithOrders(DateTime orderDate)
{
    return _session.CreateQuery("select distinct c from Customer c ," +
        " + "c.Orders.elements o where o.OrderDate > :orderDate")
        .SetDateTime("orderDate", orderDate)
        .List<Customer>();
}
```

这里使用基于面向对象的 HQL，一目了然，符合面向对象编程习惯。

写个测试用例测试 UseHQL_GetCustomersWithOrdersTest()查询方法是否正确:

```
[Test]
public void UseHQL_GetCustomersWithOrdersTest()
{
    IList<Customer> customers = _relation.UseHQL_GetCustomersWithOrders(new DateTime(2008, 10, 1));
    foreach (Customer c in customers)
    {
        foreach (Order o in c.Orders)
        {
            Assert.GreaterOrEqual(o.OrderDate, new DateTime(2008, 10, 1));
        }
    }
    foreach (Customer c in customers)
    {
        Assert.AreEqual(1, customers.Count<Customer>(x => x == c));
    }
}
```

首先调用 UseHQL_GetCustomersWithOrders()方法查询订单在 2008 年 10 月 1 号之后的顾客列表，遍历顾客列表，断言顾客为预期的 1 个，他的订单时间在 2008 年 10 月 1 号之后。OK! 测试成功。注意：这个测试用例可测试本篇所有的关联查询。

3.Criteria API 关联查询

我们使用 CreateCriteria()在关联之间导航，很容易地在实体之间指定约束。这里第二个 CreateCriteria()返回一个 ICriteria 的新实例，并指向 Orders 实体的元素。在查询中子对象使用子 CreateCriteria 语句，这是因为实体之间的关联我们在映射文件中已经定义好了。还有一种方法使用 CreateAlias()不会创建 ICriteria 的新实例。

这个例子返回顾客列表有重复的，不是我们想要的结果。

```
public IList<Customer> UseCriteriaAPI_GetCustomersWithOrders(DateTime orderDate)
{
    return _session.CreateCriteria(typeof(Customer))
        .CreateCriteria("Orders")
        .Add(Restrictions.Gt("OrderDate", orderDate))
}
```

```
.List<Customer>();  
}
```

预过滤

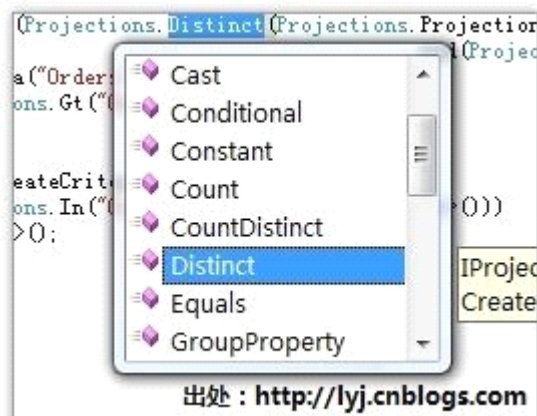
使用 ICriteria 接口的 SetResultTransformer(IResultTransformer resultTransformer)方法返回满足特定条件的 Customer。上面例子中使用条件查询，观察其生成的 SQL 语句并没有 distinct，这时可以使用 NHibernate.Transform 命名空间中的方法或者使用 NHibernate 提供的 NHibernate.CriteriaUtil.RootEntity、NHibernate.CriteriaUtil.DistinctRootEntity、NHibernate.CriteriaUtil.AliasToEntityMap 静态方法实现预过滤的作用。那么上面的查询应该修改为：

```
public IList<Customer> UseCriteriaAPI_GetCustomersWithOrders(DateTime orderDate)  
{  
    return _session.CreateCriteria(typeof(Customer))  
        .CreateCriteria("Orders")  
        .Add(Restrictions.Gt("OrderDate", orderDate))  
        .SetResultTransformer(new NHibernate.Transform.DistinctRootEntityResultTransformer())  
        //或者.SetResultTransformer(NHibernate.CriteriaUtil.DistinctRootEntity)  
        .List<Customer>();  
}
```

这个例子从转换结果集的角度实现了我们想要的效果。

投影

调用 SetProjection()方法可以实现应用投影到一个查询中。NHibernate.Criterion.Projections 是 Projection 的实例工厂，Projections 提供了非常多的方法，看看下面的截图，下拉列表中的方法是不是很多啊：



现在可以条件查询提供的投影来完成上面同样的目的：

```
public IList<Customer> UseCriteriaAPI_GetDistinctCustomers
(DateTime orderDate)
{
    IList<int> ids = _session.CreateCriteria(typeof(Customer))
        .SetProjection(Projections.Distinct(Projections.Projecti
onList()
                                .Add
(Projections.Property("CustomerId")))
        )
        .CreateCriteria("Orders")
        .Add(Restrictions.Gt("OrderDate", orderDate))
        .List<int>();

    return _session.CreateCriteria(typeof(Customer))
        .Add(Restrictions.In("CustomerId", ids.ToArray<int>
()))
        .List<Customer>();
}
```

我们可以添加若干的投影到投影列表中，例如这个例子我添加一个 `CustomerId` 属性值到投影列表中，这个列表中的所有属性值都设置了 `Distinct` 投影，第一句返回订单时间在 `orderDate` 之后所有顾客 `Distinct` 的 `CustomerId`，第二句根据返回的 `CustomerId` 查询顾客列表。达到上面的目的。这时发现其生成的 SQL 语句中有 `distinct`。我们使用投影可以很容易的组合我们需要的各种方法。

结语

这一篇通过上一篇完成的一对多关系映射，使用 `NHibernate` 中提供的三种查询方法实现了父子关联查询，并初步探讨了条件查询中比较深入的话题。希望你有所帮助。下一篇开始讨论 `NHibernate` 中的多对多映射关系和查询。

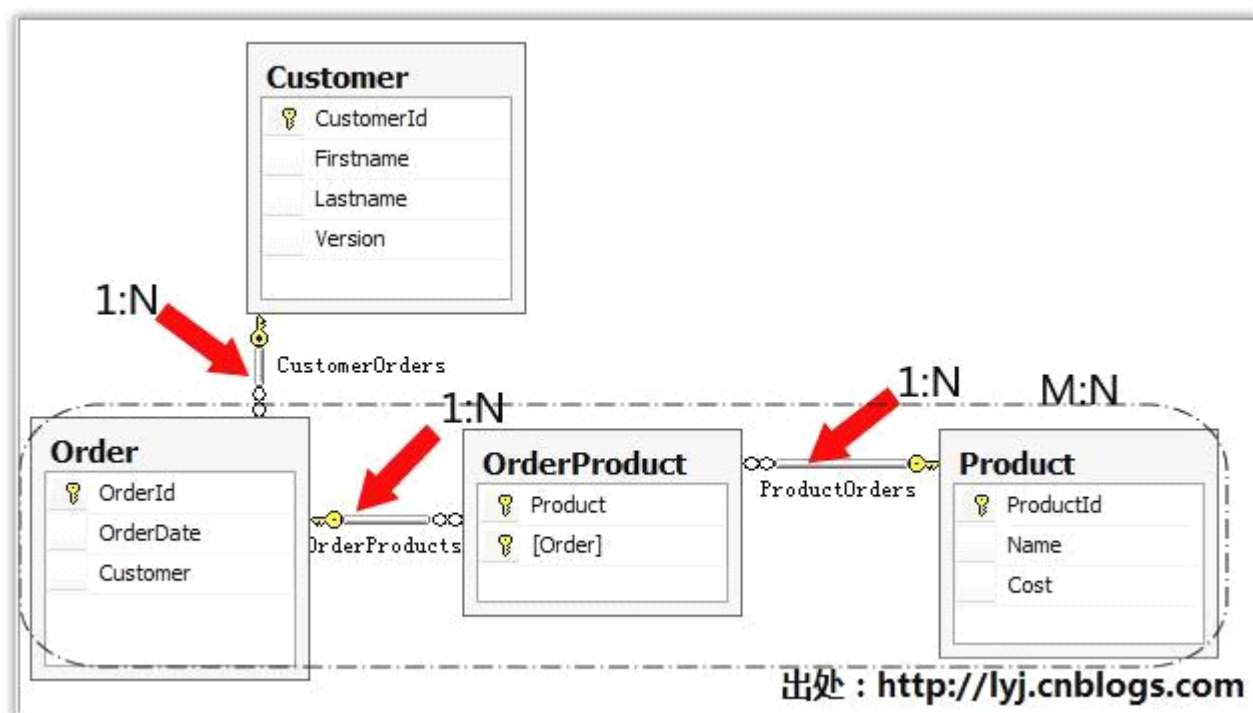
NHibernate 之旅(11): 探索多对多关系及其关联查询

本节内容

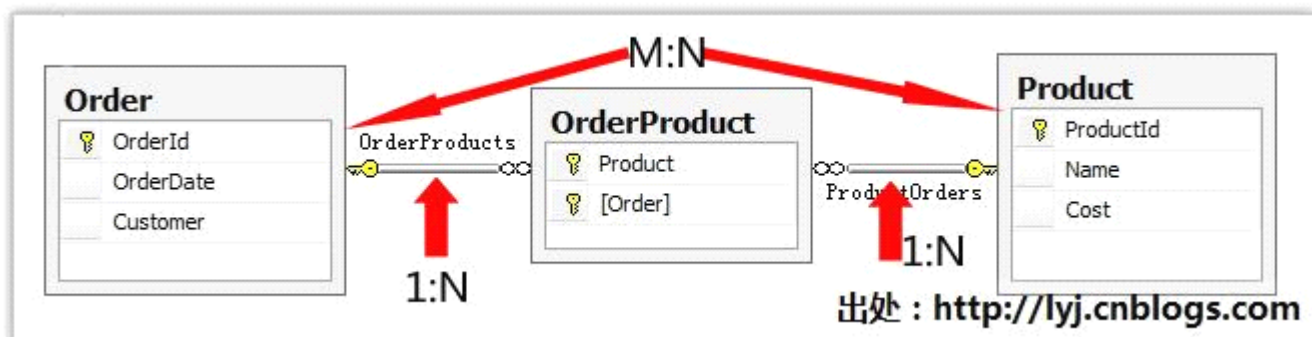
- 多对多关系引入
- 多对多映射关系
- 多对多关联查询
 - 1.原生 SQL 关联查询
 - 2.HQL 关联查询
 - 3.Criteria API 关联查询
- 结语

多对多关系引入

让我们再次回顾在第二篇中建立的数据模型:



在图上，我已经清晰的标注了表之间的关系，上两篇分析 Customer 和 Order 之间的“外键关系”或者称作“父子关系”、“一对多关系”和关联查询，这一篇以 Order 为中心，分析 Order 和 Product 之间的关系，直接看下面一幅图的两张表：



上面两张表关系表达的意思是：Order 有多个 Products，Product 属于多个 Orders。我们称 Order 和 Product 是多对多关系，这一篇我们来深入讨论在 NHibernate 如何映射多对多关系及其多对多关联查询。

多对多映射关系

1.Order 有多个 Products

有了上两篇的基础，现在直奔主题，建立关系。大家可以把这篇的代码作为模板，以后在工作中参考。

修改 Order.cs 类代码如下：

```
namespace DomainModel.Entities
{
    public class Order
    {
        public virtual int OrderId { get; set; }
        public virtual DateTime OrderDate { get; set; }
        //多对一关系: Orders 属于一个 Customer
        public virtual Customer Customer { get; set; }
        //多对多关系: Order 有多个 Products
        public virtual IList<Product> Products { get; set; }
    }
}
```

修改 Order.hbm.xml 映射文件如下：

```

<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                    assembly="DomainModel" namespace="Do
mainModel">

    <class name="DomainModel.Entities.Order,DomainModel" ta
ble="`Order`" >

        <id name="OrderId" column="OrderId" type="Int32" uns
aved-value="0">
            <generator class="native" />
        </id>
        <property name="OrderDate" column="OrderDate" type=
"DateTime" not-null="true" />
        <!--多对一关系: Orders 属于一个 Customer-->
        <many-to-one name="Customer" column="Customer" not
-null="true"
                    class="DomainModel.Entities.Customer,Doma
inModel"
                    foreign-key="FK_CustomerOrders" />
        <!--多对多关系: Order 有多个 Products-->
        <bag name="Products" generic="true" table="Orde
rProduct">
            <key column="`Order`" foreign-key="FK_Orde
rProducts"/>
            <many-to-many column="Product"
                    class="DomainModel.Entities.Produc
t,DomainModel"
                    foreign-key="FK_ProductOrders"/>
        </bag>
    </class>
</hibernate-mapping>

```

在多对多关系中，其两方都使用 Bag 集合和 many-to-many 元素。看看上面各个属性和 one-to-many, many-to-one 属性差不多。

2.Product 属于多个 Orders

在项目 DomainModel 层的 Entities 文件夹中新建 Product.cs 类，编写代码如下：


```
</bag>
</class>
</hibernate-mapping>
```

多对多关联查询

使用 NHibernate 中提供的三种查询方法实现多对多关联查询，查询返回所有订单和产品的顾客列表。

1.原生 SQL 关联查询

```
public IList<Customer> UseSQL_GetCustomersWithOrdersHavingProduct(DateTime orderDate)
{
    return _session.CreateSQLQuery("select distinct {customer.*} from Customer {customer}" +
        " inner join [Order] o on o.Customer={customer}.CustomerId"+
        " inner join OrderProduct op on o.OrderId=op.[Order]" +
        " inner join Product p on op.Product=p.ProductId where o.OrderDate > :orderDate")
        .AddEntity("customer", typeof(Customer))
        .SetDateTime("orderDate", orderDate)
        .List<Customer>();
}
```

这里需要使用 Join 告诉查询如何在表之间关联。无论多么复杂的关系，我们必须在查询语句中指定返回值。这里使用 AddEntity 设置返回的实体。

2.HQL 关联查询

```
public IList<Customer> UseHQL_GetCustomersWithOrdersHavingProduct(DateTime orderDate)
{
    return _session.CreateQuery("select distinct c from Customer c," +
        " c.Orders.elements o where o.OrderDate > :orderDate")
        .SetDateTime("orderDate", orderDate)
```

```
.List<Customer>();  
}
```

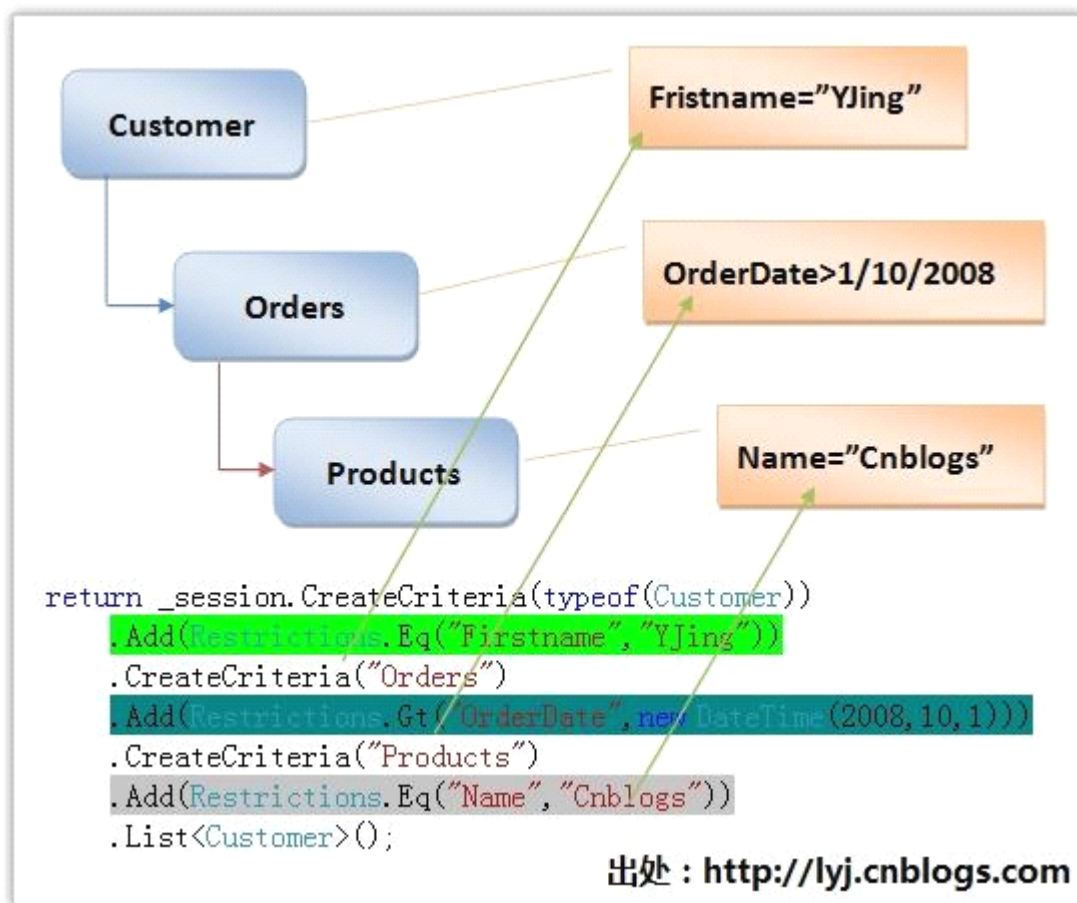
因为在映射文件已经定义实体之间一对多、多对多关系，NHibernate 通过映射文件知道如何去关联这些实体，我们不需要在查询语句中重复定义。这里使用查询和上一篇使用 HQL 关联查询语句一样，NHibernate 完全可以去关联对象，实现查询订单和产品。

3.Criteria API 关联查询

因为实体之间的关联我们在映射文件中已经定义好了。所以我们在查询子对象使用子 CreateCriteria 语句关联对象之间导航，可以很容易地在实体之间指定约束。这里第二个 CreateCriteria() 返回 ICriteria 的新实例，并指向 Orders 实体的元素。第三个指向 Products 实体的元素。

```
public IList<Customer> UseCriteriaAPI_GetCustomerswithOrdersHavingProduct()  
{  
    return _session.CreateCriteria(typeof(Customer))  
        .Add(Restrictions.Eq("Firstname","YJing"))  
        .CreateCriteria("Orders")  
        .Add(Restrictions.Gt("OrderDate",new DateTime(2008,10,1)))  
        .CreateCriteria("Products")  
        .Add(Restrictions.Eq("Name","Cnblogs"))  
        .List<Customer>();  
}
```

下面我用一幅图简单明了的说明这段代码神秘之处，也显示了一些约束条件。



编写一个测试用例测试 UseCriteriaAPI_GetCustomerswithOrdersHavingProduct() 方法，遍历列表，看看产品名称是否为"Cnblogs"，OK! 测试通过。

```

[Test]
public void UseCriteriaAPI_GetCustomerswithOrdersHavingProductTest()
{
    IList<Customer> customers = _relation.UseCriteriaAPI_GetCustomerswithOrdersHavingProduct();
    bool found = false;
    foreach (Customer c in customers)
    {
        foreach (Order o in c.Orders)
        {
            foreach (Product p in o.Products)
            {
                if (p.Name == "Cnblogs")
                {

```


NHibernate 之旅(12): 初探延迟加载机制

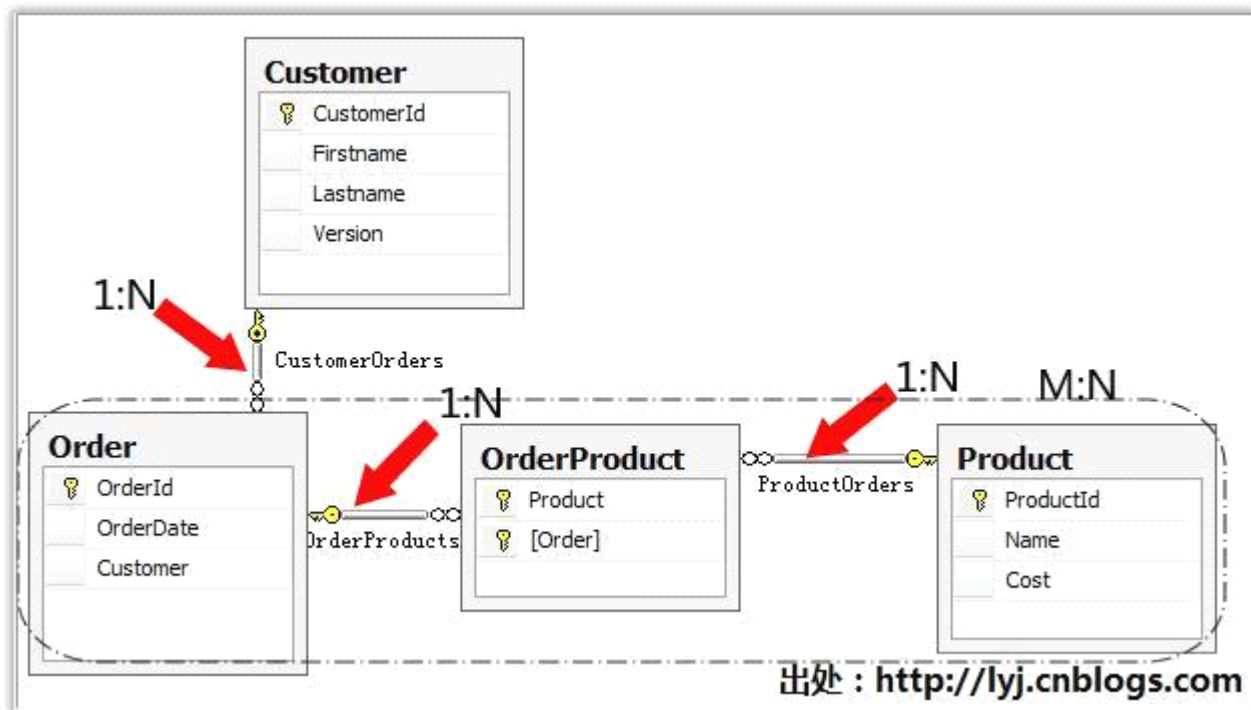
本节内容

- 引入
- 延迟加载
- 实例分析
 - 1. 一对多关系实例
 - 2. 多对多关系实例
- 结语

引入

通过前面文章的分析，我们知道了如何使用 NHibernate，比如 CRUD 操作、事务、一对多、多对多映射等问题，这篇我们初步探索 NHibernate 中的加载机制。

在讨论之前，我们看看我们使用的数据模型，回顾一下第二篇建立的数据模型。

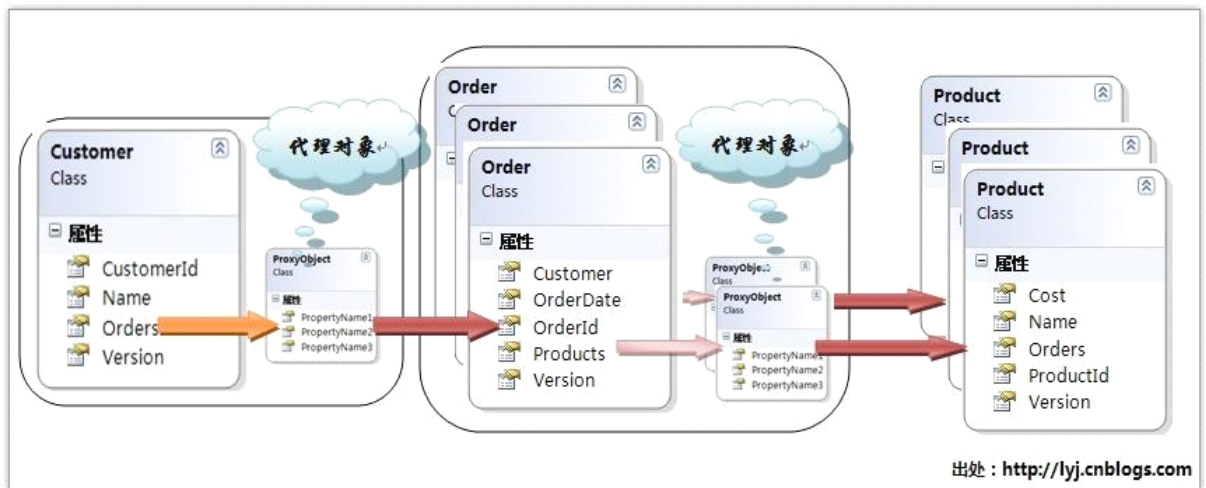


Customer 与 Orders 是一对多关系，Order 与 Product 是多对多关系。这一篇还是使用这个模型，有关具体配置和映射参考本系列的文章。

延迟加载(Lazy Loading)

延迟加载按我现在的理解应该叫“视需要加载(load-on-demand)”，“(delayed loading)”，“刚好及时加载(just-in-time loading)”在合适不过了。这里按字面理解延迟仿佛变成了“延迟，延长，拖延时间”的意思。

NHibernate 从 1.2 版本就默认支持了延迟加载。其实延迟加载的执行方式是使用 GoF23 中的代理模式，我们用一张图片来大致展示延迟加载机制。



实例分析

1. 一对多关系实例

在一对多关系实例中，我们使用 Customer 对象与 Order 对象为例，在数据访问层中编写两个方法用于在测试时调用，分别是：

数据访问层中方法一：加载 Customer 对象

```
public Customer LazyLoad(int customerId)
{
    return _session.Get<Customer>(customerId);
}
```

数据访问层中方法二：加载 Customer 对象并使用 Using 强制清理关闭 Session

```
public Customer LazyLoadUsingSession(int customerId)
```

```
{
    using (ISession _session = new SessionManager().GetSession())
    {
        return _session.Get<Customer>(customerId);
    }
}
```

1.默认延迟加载

调用数据访问层中的 LazyLoad 方法加载一个 Customer 对象，NHibernate 的默认延迟加载 Customer 关联的 Order 对象。利用 NHibernate 提供有用类(NHibernateUtil)测试被关联的 Customer 对象集合是否已初始化(也就是已加载)。

```
[Test]
public void LazyLoadTest()
{
    Customer customer = _relation.LazyLoad(1);
    Assert.IsFalse(NHibernateUtil.IsInitialized(customer.Orders));
}
```

测试成功，观察 NHibernate 生成 SQL 语句为一条查询 Customer 对象的语句。我们使用调试发现，Orders 对象集合的属性值为：{Iesi.Collections.Generic.HashSet`1[DomainModel.Entities.Order]}，并可以同时看到 Order 对象集合中的项。截图如下：

```
[Test]
public void LazyLoadTest()
{
    Customer customer = relation.LazyLoad(1);
    Assert.IsFalse(customer.Orders);
}
```

出处：<http://lyj.cnblogs.com>

2.延迟加载并关闭 Session

同第一个测试相同，这个测试调用使用 Using 强制资源清理 Session 加载 Customer 对象的方法。

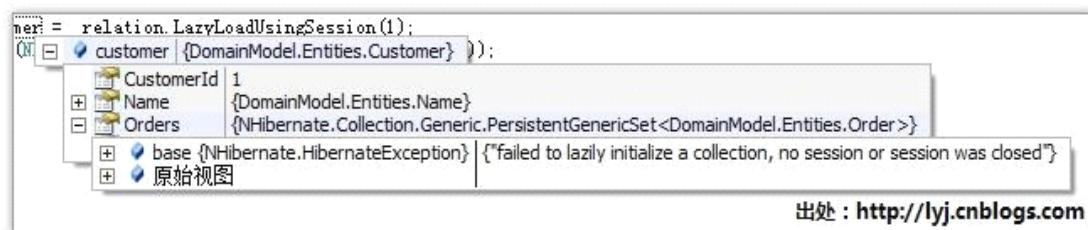
```
[Test]
```

```

public void LazyLoadUsingSessionTest()
{
    Customer customer = _relation.LazyLoadUsingSession(1);
    Assert.IsFalse(NHibernateUtil.IsInitialized(customer.Orders));
}

```

测试成功，其生成 SQL 语句和上面测试生成 SQL 语句相同。但是使用调试发现，Orders 对象集合的属性值为：NHibernate.Collection.Generic.PersistentGenericSet<DomainModel.Entities.Order>，如果你进一步想看看 Order 对象集合中的项，它抛出了 HibernateException 异常：failed to lazily initialize a collection, no session or session was closed。截图如下：



2. 多对多关系实例

同理，在多对多关系实例中，我们以 Order 对象与 Products 对象为例，我们在数据访问层中写两个方法用于测试：

方法 1：加载 Order 对象

```

public DomainModel.Entities.Order LazyLoadOrderAggregate(int orderId)
{
    return _session.Get<DomainModel.Entities.Order>(orderId);
}

```

方法 2：加载 Customer 对象并使用 Using 强制清理关闭 Session

```

public DomainModel.Entities.Order LazyLoadOrderAggregateUsingSession(int orderId)
{
    using (ISession session = new SessionManager().GetSession())

```



```

    {
        return _session.Get<DomainModel.Entities.Order>(orderId);
    }
}

```

1. 默认延迟加载

调用数据访问层中的 `LazyLoadOrderAggregate` 方法加载一个 `Order` 对象，NHibernate 的默认延迟加载 `Order` 关联的 `Products` 对象集合(多对多关系)，利用代理模式加载 `Customer` 对象集合(多对一关系)。利用 NHibernate 提供有用类(NHibernateUtil) 测试被关联的 `Products` 对象集合和 `Customer` 对象集合是否已初始化。

```

[Test]
public void LazyLoadOrderAggregateTest()
{
    Order order = _relation.LazyLoadOrderAggregate(2);
    Assert.IsFalse(NHibernateUtil.IsInitialized(order.Customer));
    Assert.IsFalse(NHibernateUtil.IsInitialized(order.Products));
}

```

测试成功，NHibernate 生成 SQL 语句如下：

```

SELECT order0_.OrderId as OrderId1_0_,
       order0_.Version as Version1_0_,
       order0_.OrderDate as OrderDate1_0_,
       order0_.Customer as Customer1_0_
FROM [Order] order0_ WHERE order0_.OrderId=@p0; @p0 =
'2'

```

调试看看效果截图，可以清楚的观察到 `Customer` 对象和 `Products` 对象集合的类型。

order = relation.LazyLoadOrderAggregate(2);
 I: order {DomainModel.Entities.Order} {der: Customer});
 Isra
 Customer {CustomerProxyac4281a97f0445d59c722ba06313eb5d}
 OrderDate {2008/10/20 0:00:00}
 OrderId 2
 Products {NHibernate.Collection.Generic.PersistentGenericBag<DomainModel.Entities.Product>}
 [0] {DomainModel.Entities.Product}
 Cost 1.0
 Name Cnblogs
 Orders {NHibernate.Collection.Generic.PersistentGenericBag<DomainModel.Entities.Order>}
 ProductId 1
 Version 1

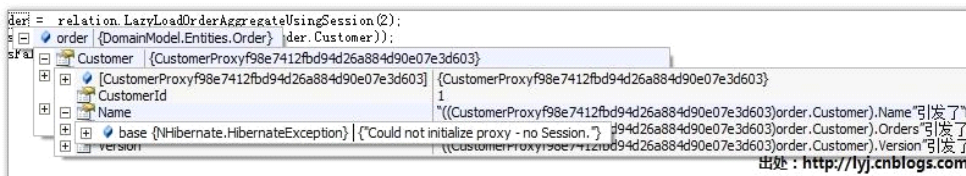
出处：<http://lyj.cnblogs.com>

2. 延迟加载并关闭 Session

同第一个测试相同，这个测试调用使用 Using 强制资源清理 Session 加载 Order 对象的方法。

```
[Test]
public void LazyLoadOrderAggregateUsingSessionTest()
{
    Order order = _relation.LazyLoadOrderAggregateUsingSession(2);
    Assert.IsFalse(NHibernateUtil.IsInitialized(order.Customer));
    Assert.IsFalse(NHibernateUtil.IsInitialized(order.Products));
}
```

测试成功，其生成 SQL 语句和上面测试生成 SQL 语句相同。但是使用调试发现，Customer 对象类型为：{CustomerProxy9dfb54eca50247f69bfedd92e1638ba5}，进一步观察 Customer 对象 Firstname、Lastname 等项引发了“NHibernate.LazyInitializationException”类型的异常。Products 对象集合的属性值为：{NHibernate.Collection.Generic.PersistentGenericBag <DomainModel.Entities.Product>}，如果你进一步想看看 Products 对象集合中的项它同样抛出 HibernateException 异常：failed to lazily initialize a collection, no session or session was closed。下面截取获取 Customer 对象一部分图，你想知道全部自己亲自调试一把：



3. 延迟加载中 LazyInitializationException 异常

上面测试已经说明了这个问题：如果我想在 Session 清理关闭之后访问 Order 对象中的某些项会得到一个异常，由于 session 关闭，NHibernate 不能为我们延迟加载 Order 项，我们编写一个测试方法验证一下：

```
[Test]
[ExpectedException(typeof(LazyInitializationException))]
public void LazyLoadOrderAggregateUsingSessionOnFailTest()
{
    Order order = _relation.LazyLoadOrderAggregateUsingSession(2);
}
```

```
string name = order.Customer.Name.Fullname;
}
```

上面的测试抛出“Could not initialize proxy - no Session”预计的 LazyInitializationException 异常，表明测试成功，证明不能加载 Order 对象的 Customer 对象。

4.N+1 选择问题

我们在加载 Order 后访问 Product 项，导致访问 Product 每项就会产生一个选择语句，我们用一个测试方法来模拟这种情况：

```
[Test]
public void LazyLoadOrderAggregateSelectBehaviorTest()
{
    Order order = _relation.LazyLoadOrderAggregate(2);
    float sum = 0.0F;
    foreach (var item in order.Products)
    {
        sum += item.Cost;
    }
    Assert.AreEqual(21.0F, sum);
}
```

NHibernate 生成 SQL 语句如下：

```
SELECT order0_.OrderId as OrderId1_0_,
       order0_.Version as Version1_0_,
       order0_.OrderDate as OrderDate1_0_,
       order0_.Customer as Customer1_0_
FROM [Order] order0_ WHERE order0_.OrderId=@p0; @p0 =
'2'

SELECT products0_.[Order] as Order1_1_,
       products0_.Product as Product1_,
       product1_.ProductId as ProductId3_0_,
       product1_.Version as Version3_0_,
       product1_.Name as Name3_0_,
       product1_.Cost as Cost3_0_
FROM OrderProduct products0_
left outer join Product product1_ on
products0_.Product=product1_.ProductId
```

```
WHERE products0_[Order]=@p0; @p0 = '2'
```

这次我走运了，NHibernate 自动生成最优化的查询语句，一口气加载了两个 Product 对象。但是试想一下有一个集合对象有 100 项，而你仅仅需要访问其中的一两项。这样加载所有项显然是资源的浪费。

幸好，NHibernate 为这些问题有一个方案，它就是立即加载。欲知事后如何，请听下回分解！

结语

这篇我们初步认识了 NHibernate 中的加载机制，这篇从一对多关系、多对多关系角度分析了 NHibernate 默认加载行为——延迟加载，下篇继续分析立即加载。希望你有所帮助。

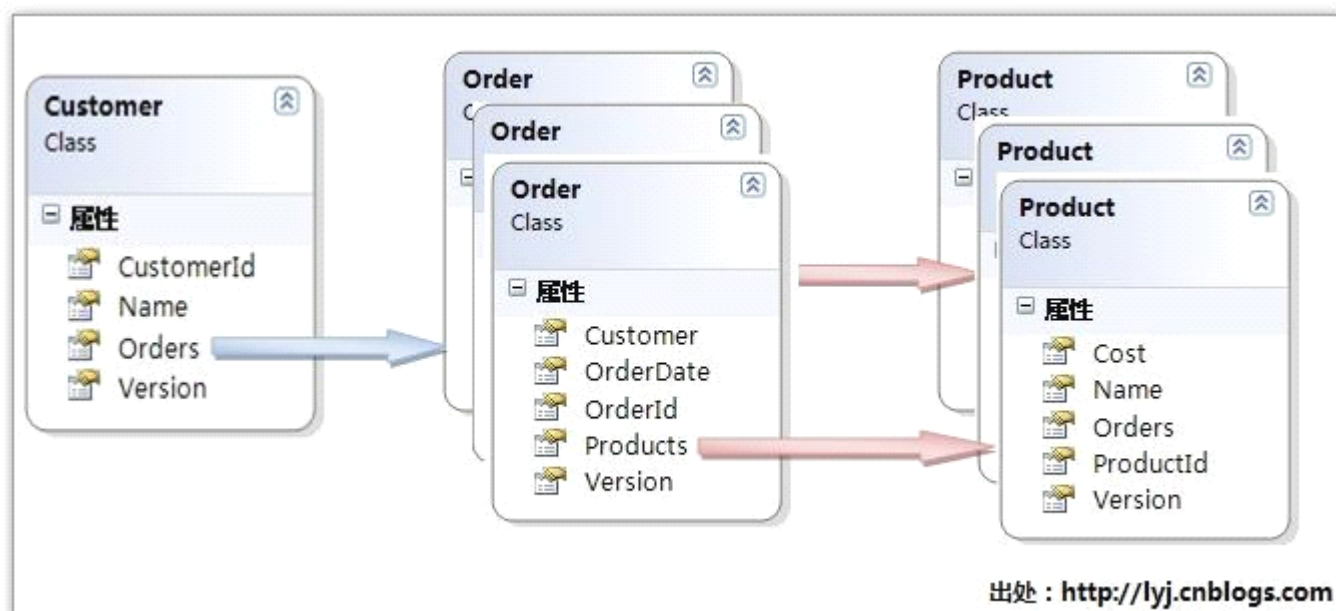
NHibernate 之旅(13): 初探立即加载机制

本节内容

- 引入
- 立即加载
- 实例分析
 - 1. 一对多关系实例
 - 2. 多对多关系实例
- 结语

引入

通过上一篇的介绍，我们知道了 NHibernate 中默认的加载机制——延迟加载。其本质就是使用 GoF23 中代理模式实现，这节我们简单分析 NHibernate 另一种加载机制——立即加载。我用一张图片形象的展现立即加载机制。



立即加载

顾名思义，就是立刻加载相关联对象集合，与延迟加载相反。我们可以使用三种方法来立即加载，分别是：可选的 `lazy` 属性、NHibernate 提供的实用类、HQL 抓取策略。下面依次用实例分析其中的机制。

实例分析

1. 一对多关系实例

在一对多关系实例中，我们使用 `Customer` 对象与 `Order` 对象为例，在数据访问层中依然使用上一篇文章的方法，这里使用强制关闭 `Session` 的方法，为什么使用 `Using` 强制释放资源呢？我就是想利用这个来模拟 `Web` 应用程序中的 `Session` 机制。用这个分析比没有 `Using` 释放资源更有意义。

数据访问层中方法：加载 `Customer` 对象并使用 `Using` 强制清理关闭 `Session`

```
public Customer LazyLoadUsingSession(int customerId)
{
    using (ISession _session = new SessionManager().GetSession())
    {
        return _session.Get<Customer>(customerId);
    }
}
```

1. 使用 `lazy="false"` 属性

在上一篇文章我们一直没有修改映射文件即一直是默认是 `lazy="true"`，NHibernate 就采用了默认的延迟加载。

这里介绍第一种方法就是修改映射文件来立即加载，打开 `Customer.hbm.xml` 文件，在 `Set` 元素中添加 `lazy="false"`。

编写一个测试验证，调用数据访问层中的使用 `Using` 强制资源清理 `Session` 加载 `Customer` 对象的方法加载一个 `Customer` 对象，NHibernate 这时立即加载 `Customer` 相关联的 `Order` 对象。利用 NHibernate 提供实用类(NHibernateUtil)测试被关联的 `Customer` 对象集合是否已初始化(也就是已加载)。

```
[Test]
public void EagerLoadUsingLazyFalseTest()
{
    Customer customer = _relation.LazyLoadUsingSession(1);
}
```

```
Assert.IsTrue(NHibernateUtil.IsInitialized(customer.Orders));
}
```

测试成功，证明 NHibernate 立即加载了 Order 对象，发现生成两句 SQL 语句：第一条查询 Customer 对象，第二条语句查询其相关联的 Order 对象集合。

```
SELECT customer0_.CustomerId as CustomerId9_0_,
       customer0_.Version as Version9_0_,
       customer0_.Firstname as Firstname9_0_,
       customer0_.Lastname as Lastname9_0_
FROM Customer customer0_ WHERE customer0_.CustomerId=
@p0; @p0 = '1'

SELECT orders0_.Customer as Customer1_,
       orders0_.OrderId as OrderId1_,
       orders0_.OrderId as OrderId6_0_,
       orders0_.Version as Version6_0_,
       orders0_.OrderDate as OrderDate6_0_,
       orders0_.Customer as Customer6_0_
FROM [Order] orders0_ WHERE orders0_.Customer=@p0; @p
0 = '1'
```

不过，细心的朋友会发现，这时 Orders 对象集合的类型是 `Iesi.Collections.Generic.HashedSet`1[DomainModel.Entities.Order]`，上一节只有在没有使用 Using 强制关闭资源下，Orders 对象集合才是这个类型，在使用强制关闭资源的情况下，Orders 对象集合的类型为：`NHibernate.Collection.Generic.PersistentGenericSet<DomainModel.Entities.Order>`，进一步读取 Order 项抛出 `HibernateException` 异常。我想从这个角度也说明了立即加载机制。

好了，这就说到这里，还是把映射文件改为原来默认的吧(即去掉 `lazy="false"`)，看看还有其它什么方法来立即加载。

2.使用 NHibernateUtil 实用类

NHibernate 提供实用类(NHibernateUtil)不光光只是用来测试被关联的对象集合是否已初始化，还有一个非常重要的功能就是**可以强制初始化未初始化的相关联的对象**。有了这个功能，我们就可以修改数据访问层中的方法，把上面使用 Using 强制清理关闭 Session 的方法中加上 NHibernateUtil 类提供 Initialize 方法来初始化 Customer 相关联的 Order 对象集合。

```

public Customer EagerLoadUsingSessionAndNHibernateUtil(int
customerId)
{
    using (ISession _session = new SessionManager().GetSess
ion())
    {
        Customer customer= _session.Get<Customer>(custo
merId);
        NHibernateUtil.Initialize(customer.Orders);
        return customer;
    }
}

```

我们编写一个方法来测试一下：

```

[Test]
public void EagerLoadUsingSessionAndNHibernateUtilTest()
{
    Customer customer = _relation.EagerLoadUsingSessionAn
dNHibernateUtil(1);
    Assert.IsTrue(NHibernateUtil.IsInitialized(customer.Order
s));
}

```

测试成功，这个结果同修改映射文件一样。

2. 多对多关系实例

1. 使用 lazy="false" 属性

同理，使用 lazy="false" 属性来设置立即加载行为，这时在持久化类中就不必为其公共方法、属性和事件声明为 virtual 属性了，因为没有使用延迟加载。不过在这里我还是推荐大家使用 NHibernate 默认的延迟加载行为，原因很简单，NHibernate 延迟加载性能上可以提高很多，在特殊情况下使用下面的方法来立即加载。

这个例子同上面类似，这里就不举重复的例子了，大家自己测试下就可以了。

2. 使用 NHibernateUtil 实用类

如果你需要获得 Order 实体的相关联对象可以使用 NHibernateUtil 类初始化关联对象(把他们从数据库取出来)。看看下面数据访问层中的方法，使用 NHibernateUtil 类提供 Initialize 方法初始化相关联的 Customer 和 Product 对象。


```

public DomainModel.Entities.Order
    EagerLoadOrderAggregateSessionAndNHibernateUtil(int orderId)
{
    using (ISession _session = new SessionManager().GetSession())
    {
        DomainModel.Entities.Order order =
            _session.Get<DomainModel.Entities.Order>(orderId);

        NHibernateUtil.Initialize(order.Customer);
        NHibernateUtil.Initialize(order.Products);
        return order;
    }
}

```

测试上面的方法:

```

[Test]
public void EagerLoadOrderAggregateSessionAndNHibernateUtilTest()
{
    Order order =
        _relation.EagerLoadOrderAggregateSessionAndNHibernateUtil(2);
    Assert.IsTrue(NHibernateUtil.IsInitialized(order.Customer));
    Assert.IsTrue(NHibernateUtil.IsInitialized(order.Products));
    Assert.AreEqual(order.Products.Count, 2);
}

```

看看 NHibernate 生成的 SQL 语句，真是多了，一对多关系，多对多关系的一次立即加载就生成了四条 SQL 语句，分别查询了 Order 表，Customer 表，OrderProduct 表相关联的 Product。(Customer 与 Order 一对多关系在这里也立即加载了一次)，这时内存中的内容都是这些关联对象的值，你也不是每个对象都用到，何必要全部加载呢。

```

SELECT order0_.OrderId as OrderId6_0_,
       order0_.Version as Version6_0_,
       order0_.OrderDate as OrderDate6_0_,
       order0_.Customer as Customer6_0_

```

```
FROM [Order] order0_ WHERE order0_.OrderId=@p0; @p0 = '2'
```

```
SELECT customer0_.CustomerId as CustomerId9_0_,  
       customer0_.Version as Version9_0_,  
       customer0_.Firstname as Firstname9_0_,  
       customer0_.Lastname as Lastname9_0_  
FROM Customer customer0_ WHERE customer0_.CustomerId=@p0; @p0 = '1'
```

```
SELECT orders0_.Customer as Customer1_,  
       orders0_.OrderId as OrderId1_,  
       orders0_.OrderId as OrderId6_0_,  
       orders0_.Version as Version6_0_,  
       orders0_.OrderDate as OrderDate6_0_,  
       orders0_.Customer as Customer6_0_  
FROM [Order] orders0_ WHERE orders0_.Customer=@p0; @p0 = '1'
```

```
SELECT products0_.[Order] as Order1_1_,  
       products0_.Product as Product1_,  
       product1_.ProductId as ProductId8_0_,  
       product1_.Version as Version8_0_,  
       product1_.Name as Name8_0_,  
       product1_.Cost as Cost8_0_  
FROM OrderProduct products0_  
left outer join Product product1_ on products0_.Product=product1_.ProductId  
WHERE products0_.[Order]=@p0; @p0 = '2'
```

3.使用 HQL 抓取策略

使用 HQL 查询方法也可以立即加载。HQL 语句支持的连接类型为：inner join(内连接)、left outer join(左外连接)、right outer join(右外连接)、full join(全连接，不常用)。

“抓取 fetch”连接允许仅仅使用一个选择语句就将相关联的对象随着他们的父对象的初始化而被初始化，可以有效的代替了映射文件中的外联接与延迟属性声明。

几点注意：

- `fetch` 不与 `setMaxResults()` 或 `setFirstResult()` 共用，因为这些操作是基于结果集的，而在预先抓取集合时可能包含重复的数据，也就是说无法预先知道精确的行数。
- `fetch` 还不能与独立的 `with` 条件一起使用。通过在一次查询中 `fetch` 多个集合，可以制造出笛卡尔积，因此请多加注意。对多对多映射来说，同时 `join fetch` 多个集合角色可能在某些情况下给出并非预期的结果，也请小心。
- 使用 `full join fetch` 与 `right join fetch` 是没有意义的。如果你使用属性级别的延迟获取，在第一个查询中可以使用 `fetch all properties` 来强制 NHibernate 立即取得那些原本需要延迟加载的属性。

下面写个简单例子说明：

```
public DomainModel.Entities.Order EagerLoadOrderAggregate
WithHQL(int orderId)
{
    using (ISession _session = new SessionManager().GetSession())
    {
        return _session.CreateQuery("from Order o"+
            " left outer join fetch o.Products" +
            " inner join fetch o.Customer where o.OrderId=:orderId")
            .SetInt32("orderId", orderId)
            .UniqueResult<DomainModel.Entities.Order>();
    }
}
```

编写测试用例测试上面的方法：验证构建一个 HQL 查询不仅加载 `Order`，也加载了相关联的 `Customer` 和 `Product` 对象。

```
[Test]
public void EagerLoadOrderAggregateWithHQLTest()
{
    Order order = _relation.EagerLoadOrderAggregateWithHQL(2);
    Assert.IsTrue(NHibernateUtil.IsInitialized(order.Customer));
    Assert.IsTrue(NHibernateUtil.IsInitialized(order.Products));
    Assert.AreEqual(order.Products.Count, 2);
}
```

```
}
```

通过 NHibernate 生成 SQL 语句可以说明 NHibernate 可以一口气立即加载 Order 和所有 Order 相关联的 Customer 和 Product 对象。SQL 语句生成如下：

```
select order0_.OrderId as OrderId6_0_,
       product2_.ProductId as ProductId8_1_,
       customer3_.CustomerId as CustomerId9_2_,
       order0_.Version as Version6_0_,
       order0_.OrderDate as OrderDate6_0_,
       order0_.Customer as Customer6_0_,
       product2_.Version as Version8_1_,
       product2_.Name as Name8_1_,
       product2_.Cost as Cost8_1_,
       customer3_.Version as Version9_2_,
       customer3_.Firstname as Firstname9_2_,
       customer3_.Lastname as Lastname9_2_,
       products1_.[Order] as Order1_0___,
       products1_.Product as Product0___
from [Order] order0_
left outer join OrderProduct products1_ on order0_.OrderId=products1_.[Order]
left outer join Product product2_ on products1_.Product=product2_.ProductId
inner join Customer customer3_ on order0_.Customer=customer3_.CustomerId
where (order0_.OrderId=@p0 ); @p0 = '2'

SELECT orders0_.Customer as Customer1_,
       orders0_.OrderId as OrderId1_,
       orders0_.OrderId as OrderId6_0_,
       orders0_.Version as Version6_0_,
       orders0_.OrderDate as OrderDate6_0_,
       orders0_.Customer as Customer6_0_
FROM [Order] orders0_ WHERE orders0_.Customer=@p0; @p0 = '1'
```

通过使用 HQL 抓取策略可以很好的在程序中编写出自己想要的结果。

结语

通过这篇和上一篇我们初步认识了 **NHibernate** 中的加载机制，依次从一对多关系、多对多关系角度分析了 **NHibernate** 默认延迟加载和立即加载。这些仅仅是我在平时应用、学习中摸索出来的一点收获，并非官方认可的东西，希望你有所帮助。

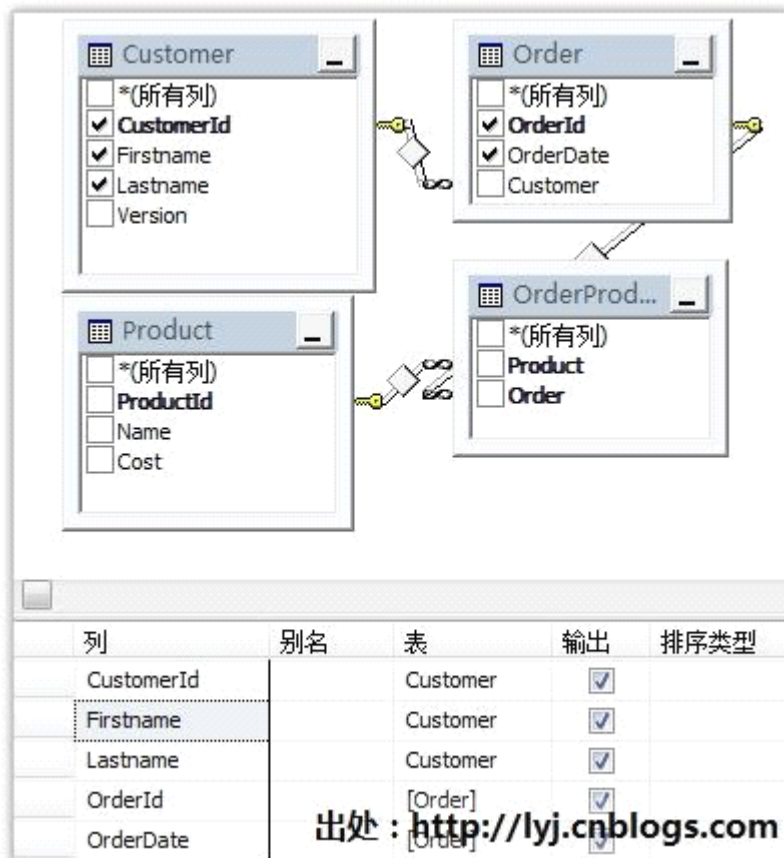
NHibernate 之旅(14): 探索 NHibernate 中使用视图

本节内容

- 引入
- 1.持久化类
- 2.映射文件
- 3.测试
- 结语

引入

在数据库操作中，我们除了对表操作，还有视图、存储过程等操作，这一篇和下篇来学习这些内容。这篇我们来学习如何在 NHibernate 中使用视图。首先，我们在数据库中建立一个名为 viewCustomer 视图，选中 CustomerId、Firstname、Lastname、OrderId、OrderDate 项。



下面我们依次为这个视图编写持久化类和映射吧。

1.持久化类

同持久化数据库中的表类似，我们需要对视图持久化，定义视图中的每个属性，因为视图是只读的，所以在这里我们只要把属性的 Setter 设置为 private 访问权限。具体做法如下：

在项目 DomainModel 层的 Entities 文件夹中新建 CustomerView.cs 类，编写代码如下：

```
namespace DomainModel.Entities
{
    public class CustomerView
    {
        public virtual int CustomerId { get; private set; }
        public virtual string Firstname { get; private set; }
        public virtual string Lastname { get; private set; }
        public virtual int OrderId { get; private set; }
        public virtual DateTime OrderDate { get; private set; }
    }
}
```

2. 映射文件

在项目 DomainModel 层的 Mappings 文件夹中新建 CustomerView.hbm.xml 文件，与映射数据库表类似，编写代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
                    assembly="DomainModel" namespace="Do
mainModel">

    <class name="DomainModel.Entities.CustomerView,Domain
Model"
          table="viewCustomer" mutable="false" >
        <id name="CustomerId" column="CustomerId" type="Int
32">
            <generator class="native" />
        </id>
        <property name="Firstname" column="Firstname" type="
string" />
        <property name="Lastname" column="Lastname" type="s
tring" />
        <property name="OrderId" column="OrderId" type="Int3
2" />
        <property name="OrderDate" column="OrderDate" type=
"DateTime" />
    </class>
</hibernate-mapping>
```

好了，到这里我们准备工作就做完了，即完成了持久化和映射。下面我们可以使用视图了。

3. 测试

在数据访问层(DAL)中编写一个方法获取订单时间在 orderDate 之后的顾客列表，方法如下：

```
public IList<CustomerView> GetCustomerView(DateTime orde
rDate)
{
    return _session.CreateCriteria(typeof(CustomerView))
```



```
.Add(Restrictions.Gt("OrderDate", orderDate))
.List<CustomerView>();
}
```

在数据访问测试层(DAL.Test)中编写一个方法由于测试上面的方法。首先调用这个方法查询出订单时间在 2008 年 10 月 1 日之后的顾客列表，断言其订单时间是否大于 2008 年 10 月 1 日。

```
[Test]
public void GetCustomerViewTest()
{
    DateTime testorderDate = new DateTime(2008, 10, 1);
    IList<CustomerView> customers =
        _relation.GetCustomerView(testorderDate);
    foreach (CustomerView view in customers)
    {
        Assert.GreaterOrEqual(view.OrderDate, testorderDate);
    }
}
```

OK! 测试通过，NHibernate 生成 SQL 语句如下：

```
SELECT this_.CustomerId as CustomerId0_0_,
       this_.Firstname as Firstname0_0_,
       this_.Lastname as Lastname0_0_,
       this_.OrderId as OrderId0_0_,
       this_.OrderDate as OrderDate0_0_
FROM viewCustomer this_
WHERE this_.OrderDate > @p0; @p0 = '2008/10/1 0:00:00'
```

好了，到此我们学会了在 NHibernate 中如何使用视图，是不是很简单啊。

结语

通过这篇文章的展示，我们学习了在 NHibernate 中如何使用视图，同表类似，只是属性访问权限不同罢了，如果你原来不知道如何使用视图，网上到现在也没有相关资料觉得无从下手，通过这篇文章的快速阅读，是不是使用视图非常简单，豁然开朗的样子(视图原来这样啊，没有什么神秘之处~~)。下篇我们来看看 NHibernate 中使用储存过程，用过存储过程的朋友都知道，真是烦人，这个存储过程我真是弄了很长时间，在实际运用中错误不断，我把它一一化解，请征集意见，大家说下篇是写一一化解的整个详细过程(涉及错误信息，如何修改，2 篇样子)还是直接讲正确方案(1 篇搞定)。由你做主！

NHibernate 之旅(15): 探索 NHibernate 中使用存储过程(上)

本节内容

- 引入
- 使用 MyGeneration 生成存储过程
- 实例分析
 - 1.删除对象
- 结语

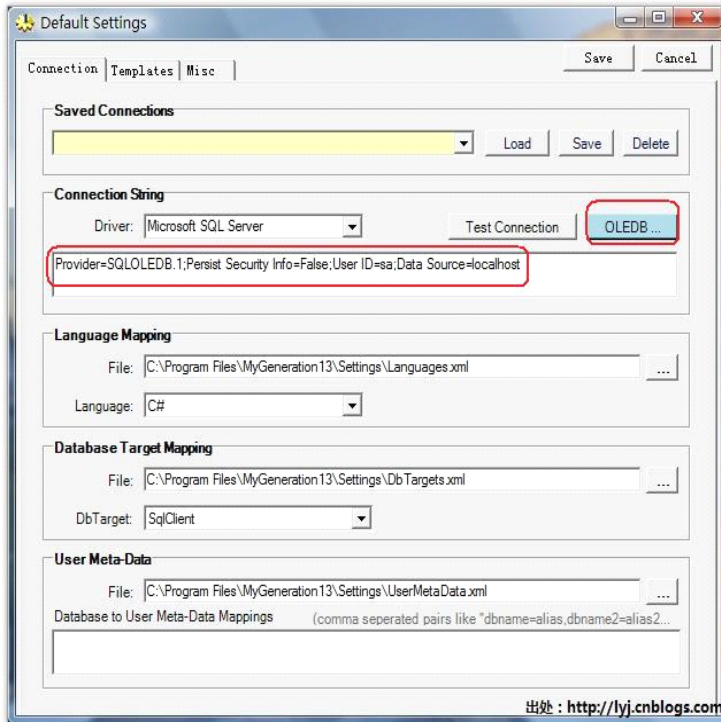
引入

上一篇，我们介绍了视图，征集大家的意见，我接下来可能用三篇篇幅来介绍在 NHibernate 中如何使用存储过程的整个详细过程，这些全是在实际运用中积累的经验，涉及刚刚使用的错误信息，如何修改存储过程，并且比较没有使用存储过程的不同点，并非官方比较权威的资料，所以敬请参考。

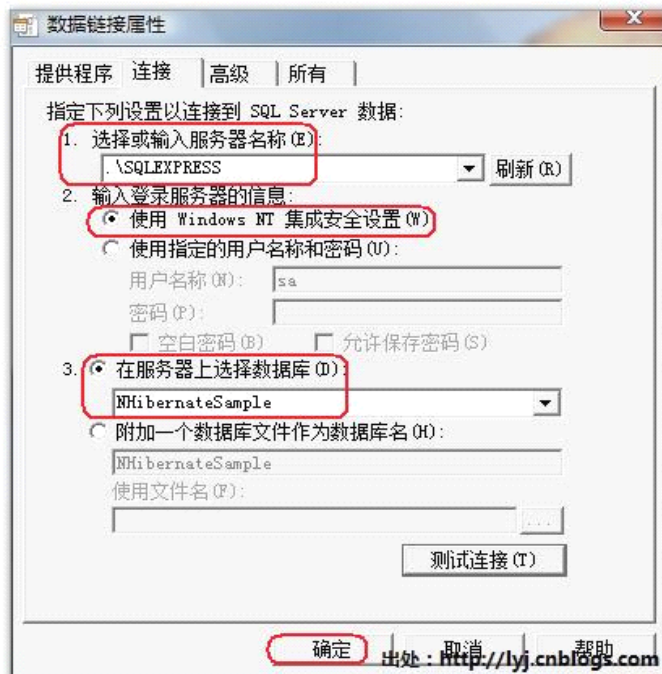
使用 MyGeneration 生成存储过程

由于写存储过程不是这节的重点，我们来利用 MyGeneration 代码生成工具来利用为 Customer 表生成插入、更新、删除记录的存储过程。顺便也来学习 MyGeneration 开源的生成器怎么用的，如果你还没有，请到[这里](#)下载安装。

安装完之后，打开 MyGeneration，如果你第一次使用 MyGeneration 会自动弹出“默认设置”对话框，需要你对 MyGeneration 设置数据库连接字符串、模板语言、数据库驱动、模板存放路径等信息，截图如下(由于这篇图片比较多，点击图片放大)：



我们要首先配置数据库连接字符串，如果你对手写感兴趣，你可以直接在文本框中修改具体字符串；如果你和我一样，就点击“OLEDB...”按钮利用界面配置连接字符串，点击按钮之后，出来“数据连接属性”对话框，中文的哦，MyGeneration 直接调用 SQL Server 的配置对话框，按下面图示，填写自己的连接字符串信息：

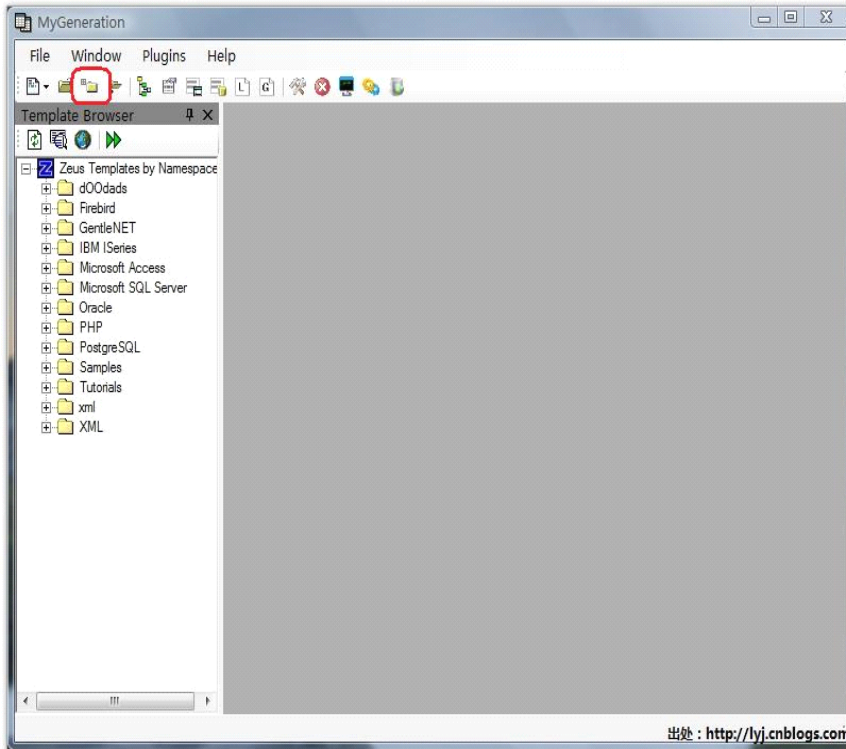


然后点击“确定”，这个对话框的连接信息将保存在“默认配置”对话框中，点击“Save”保存配置信息。

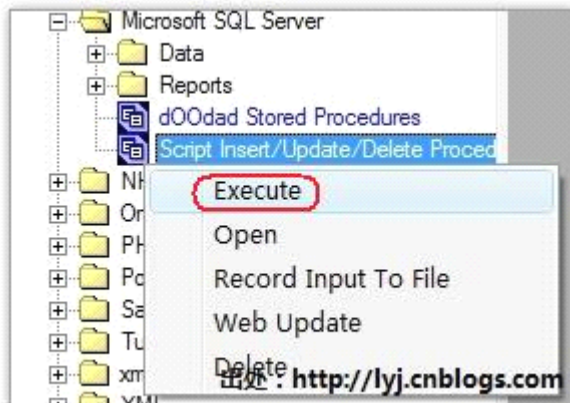
MyGeneration 使用提示

如果你的操作系统是 Windows Vista，请右击“以管理员身份运行”MyGeneration，这一点我还是比较郁闷的，弄了很久 MyGeneration 配置信息和模板保存失败，后来换成以管理员身份运行问题全部解决。

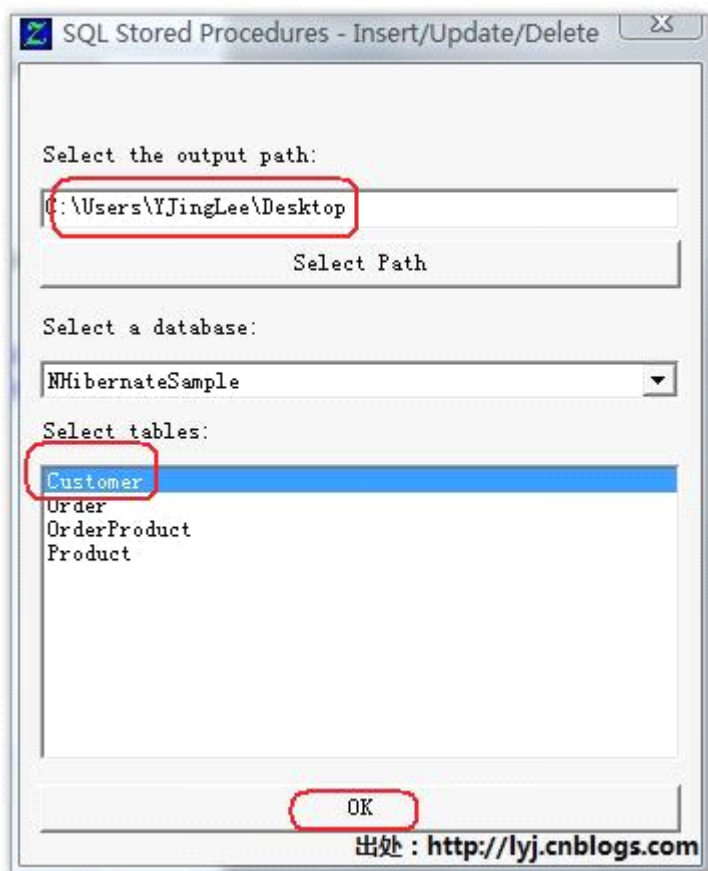
这时，MyGeneration 主界面就打开了，点击工具栏的第三个按钮，打开“模板浏览器”窗口，截图如下：



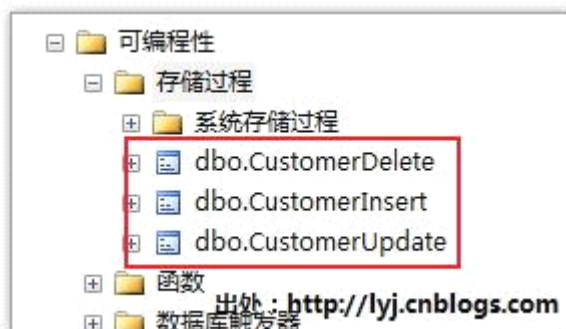
展开 Microsoft SQL Server 节点，找到“Script Insert/Update/Delete Procedures for SQL Server”模板，右击选择执行，我们利用这个模板为 Customer 表生成增删改存储过程。



打开后，这个模板界面如下，选择输出路径和数据库表，这里我输入路径为桌面，选择 Customer 表，点击 OK。截图如下：



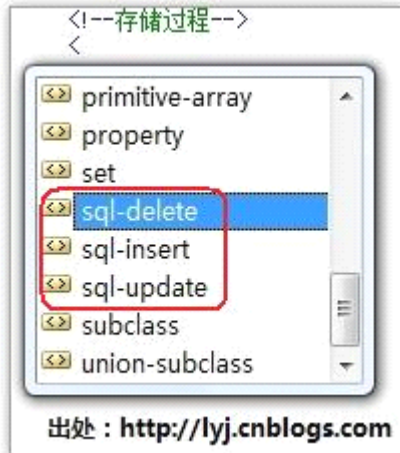
这时在桌面上生成 sql_procs_Customer.sql 文件，打开 SQL Server Management Studio 执行这个 SQL 脚本，展开 NHibernateSample 表下的存储过程，已经出来了三个存储过程，分别是 CustomerDelete、CustomerInsert、CustomerUpdate。



好了，我们使用 MyGeneration 生成存储过程，这个工作完成了，但是噩梦开始了.....

实例分析

还是一步一步来，依次从删除对象、新建对象、更新对象、查询对象来介绍在 NHibernate 中如何使用存储过程的整个详细过程。在 NHibernate 的映射文件中，在 Class 元素中提供了<sql-delete>、<sql-insert>、<sql-update>元素用于删除、新建、更新对象，注意这三个元素顺序唯一，就是下图显示的顺序，在根元素提供了<sql-query>元素用来查询对象，下图显示在 Class 元素中的增删改存储过程元素。



1.删除对象

这篇数据访问层中代码我们使用 [NHibernate 之旅\(6\): 探索 NHibernate 中的事务](#) 中的代码，测试代码在 [NHibernate 之旅\(5\): 探索 Insert, Update, Delete 操作](#) 中体现，无需任何改动。

Step1: 为了比较，我们先**执行 DeleteCustomerTest()测试方法**，没有使用存储过程下删除对象生成 SQL 语句如下：

```
DELETE FROM Customer WHERE CustomerId = @p0 AND Version = @p1; @p0 = '7', @p1 = '1'
```

Step2: **修改映射文件添加存储过程**，打开 Customer.hbm.xml 映射文件，在 Class 元素下添加<sql-delete>节点，调用 CustomerDelete 存储过程，CustomerDelete 存储过程有一个 CustomerId 参数，这里用一个问号表示：

```
<sql-delete>exec CustomerDelete ?</sql-delete>
```

Step3: 执行 DeleteCustomerTest()测试方法，出现错误“NHibernate.StaleObjectStateException : Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect): [DomainModel.Entities.Customer#4]”，这个错误是存储过程写法错误，我们**修改 CustomerDelete 存储过程**，去掉 SET NOCOUNT ON，代码片段如下：

```
ALTER PROCEDURE [dbo].[CustomerDelete]
```

```

(
    @CustomerId int
)
AS
--SET NOCOUNT ON
DELETE
FROM [Customer]
WHERE
    [CustomerId] = @CustomerId
RETURN @@Error

```

Step4: 执行 **DeleteCustomerTest()** 测试成功，NHibernate 生成语句如下：

```
exec CustomerDelete @p0; @p0 = '6', @p1 = '1'
```

恩，成功了，但是在看看存储过程，想想存储过程中写的 SQL 语句，和没有使用存储过程，NHibernate 生成的 SQL 语句有些不同，显然没有顾及到版本控制问题，使用存储过程 NHibernate 生成语句无缘无故增加了一个参数 @p1，我们来解决一下：

Step5: **修改存储过程添加 Version 处理：**

```

ALTER PROCEDURE [dbo].[CustomerDelete]
(
    @CustomerId int,
    @Version int
)
AS
DELETE
FROM [Customer]
WHERE
    [CustomerId] = @CustomerId and [Version] = @Version
RETURN @@Error

```

Step6: 执行 DeleteCustomerTest() 测试，发生错误：“过程或函数 'CustomerDelete' 需要参数 '@Version'，但未提供该参数。”哦，原来映射文件参数数量没有改。

Step7: **修改存储过程参数数量**，打开映射文件在 <sql-delete> 中添加一个参数即添加“ ,?”

```
<sql-delete>exec CustomerDelete ?,?</sql-delete>
```

Step8: 最后执行 **DeleteCustomerTest()** 测试成功，NHibernate 生成 SQL 语句如下，和 NHibernate 没有使用存储过程一样了，考虑了版本控制问题，完全符合。OK!!

```
exec CustomerDelete @p0,@p1; @p0 = '8', @p1 = '1'
```

当然了，如果你不想使用存储过程，也可以直接在<sql-delete>中写 SQL 语句，像这样，照样用。

```
<sql-delete>DELETE FROM [Customer] WHERE [CustomerId]  
= ? and [Version] =?</sql-delete>
```

结语

这篇就说到这里了，主要学习怎么使用 MyGeneration 提供的模板创建存储过程和删除对象存储过程的使用，下篇继续创建、更新、查询对象存储过程的使用。

NHibernate 之旅(16): 探索 NHibernate 中使用存储过程(中)

本节内容

- 引入
- 实例分析
 - 2.创建对象
 - 3.更新对象
- 结语

引入

上一篇，怎么使用 MyGeneration 提供的模板创建存储过程和删除对象存储过程的使用，这篇接下来介绍在 NHibernate 中如何使用存储过程创建对象、更新对象整个详细过程，这些全是在实际运用中积累的经验，涉及使用的错误信息，如何修改存储过程，并且比较没有使用存储过程的不同点，并非官方比较权威的资料，所以敬请参考，这篇继续，如果你还没有来得及看[上一篇](#)，那赶紧去看看吧。

实例分析

2.创建对象

Step1: 为了比较，我们先**执行 CreateCustomerTest() 测试方法**，没有使用存储过程下创建对象生成 SQL 语句如下：

```
INSERT INTO Customer (Version, Firstname, Lastname) VALUE
S (@p0, @p1, @p2);
select SCOPE_IDENTITY(); @p0 = '1', @p1 = 'YJing', @p2 = 'L
ee'
```

Step2: **修改映射文件添加存储过程**，打开 Customer.hbm.xml 映射文件，在 Class 元素下添加<sql-insert>节点，调用 CustomerInsert 存储过程，CustomerInsert 存储过程有四个参数，这里用四个问号表示：

```
<sql-insert>exec CustomerInsert ?,?,?,?</sql-insert>
```

Step3: 执行 CreateCustomerTest()测试方法, 出现错误“NHibernate.Exceptions.GenericADOException : could not insert: [DomainModel.Entities.Customer][SQL: exec CustomerInsert ?,?,?,?]; System.Data.SqlClient.SqlException : 参数化查询 '@p0 int,@p1 nvarchar(3),@p2 nvarchar(7),@p3 int)exec CustomerIn' 需要参数 '@p3', 但未提供该参数”, 这应该是参数问题, 仔细看看原来的存储过程, 参数位置有些问题。

Step4: **修改 CustomerInsert 存储过程**, 去掉 SET NOCOUNT ON 并把参数移动位置, 代码片段如下:

```
ALTER PROCEDURE [dbo].[CustomerInsert]
(
    @Version int,
    @Firstname nvarchar(50) = NULL,
    @Lastname nvarchar(50) = NULL,
    @CustomerId int = NULL OUTPUT
)
AS
INSERT INTO [Customer]
(
    [Version],
    [Firstname],
    [Lastname]
)
VALUES
(
    @Version,
    @Firstname,
    @Lastname
)
SELECT @CustomerId = SCOPE_IDENTITY();
RETURN @@Error
```

Step4: 执行 **CreateCustomerTest()测试方法失败**, 还是以上问题, 是不是生成器问题?

这里, 先看看 NHibernate 中最常用的两个内置生成器:

native: 根据底层数据库的能力选择 identity、sequence 或者 hilo 中的一个。

increment: 生成类型为 Int64、Int16 或 Int32 的标识符, 只当没有其他进程同时往同一个表插入数据时, 能够保持唯一性。

附：

- **identity**：对 DB2、MySQL、SQL Server、Sybase 数据库内置标识字段提供支持。数据库返回的标识符用 `Convert.ChangeType` 加以转换，因此能够支持任何整型。
- **sequence**：在 DB2、PostgreSQL、Oracle 数据库中使用序列或者 Firebird 的生成器。数据库返回的标识符用 `Convert.ChangeType` 加以转换，因此能够支持任何整型。
- **hilo**：使用一个高/低位算法高效地生成 `Int64`、`Int32` 或 `Int16` 类型的标识符。给定一个表和字段（默认分别是 `hibernate_unique_key` 和 `next_hi`）作为高位值的来源。高/低位算法生成的标识符只在一个特定的数据库中是唯一的。如果是用户提供的连接，不要使用此生成器。

测试上面方法时，使用 NHibernate 内置生成器类型 `native`，它根据数据库配置自动选择了 `identity` 类型，`identity` 类型对表的主键提供支持，可以为主键插入显式值(标识增量加一)。我们在第一步就是使用 NHibernate 内置的生成器类型为主键增量加一，没有任何问题，但是看看 `CustomerInsert` 存储过程，主键 `CustomerId` 使用 `SCOPE_IDENTITY()` 插入显式值(标识增量加一)，我们就没有必要使用内置的生成器来插入值了，把设置 `IDENTITY_INSERT` 为 `OFF`，NHibernate 正好提供了 `increment` 类型，生成类型仅仅是整型。

解决方法有两种：

- **1.修改存储过程**：如果你在开发，你最好修改存储过程，因为在面向对象开发中，尽量不要使用存储过程，何况现在存储过程破坏了你的设计。
- **2.修改主键生成类型**：如果你已经部署好你的数据库，你没有权限修改存储过程的话，那么只要修改程序来将就存储过程了。

还有一点注意：如果你主键生成器类型为“`native`”，那么存储过程的参数必须相一致。

Step5: 修改主键生成器类型

为了演示，这里我们修改主键生成器类型，还可以总结错误信息。使用 `increment` 类型，关闭 `IDENTITY_INSERT`，这时执行存储过程，由存储过程来为表'`Customer`'中的标识列(主键)插入显式值(标识增量加一)。

```
<generator class = "increment" ></generator>
```

Step6: 执行 `CreateCustomerTest()` 测试方法成功，生成 SQL 如下，其中 `p0` 是 `Version`，`p3` 是 `CustomerId`

```
exec CustomerInsert @p0,@p1,@p2,@p3;
```

```
@p0 = '1', @p1 = 'YJing', @p2 = 'Lee', @p3 = '18'
```

错误提示

其实我在调试过程中还有一些错误，这里总结一下：

方案 1：使用主键生成器类型为"native"

- 直接创建对象：正常
- 存储过程创建对象：参数化查询 '(@p0 int,@p1 nvarchar(5),@p2 nvarchar(7),@p3 int)exec CustomerIn' 需要参数 '@p3'，但未提供该参数。解决方法：使用 increment 类型

方案 2：使用主键生成器类型为"increment"

- 直接创建对象：当IDENTITY_INSERT 设置为 OFF 时，不能为表'Customer'中的标识列插入显式值。解决方法：使用 native 类型
- 存储过程创建对象：Batch update returned unexpected row count from update; actual row count: -1; expected: 1。解决方法：去掉 SET NOCOUNT ON

另外，如果你不喜欢存储过程的话，你也可以这样写，效果和使用存储过程一样。

```
<sql-insert>INSERT INTO Customer (Version, Firstname, Last name) VALUES (?, ?, ?)</sql-insert>
```

但是这样的话，主键生成器类型必须为"increment"。

3.更新对象

Step1: 为了比较，我们先**执行 UpdateCustomerTest()测试方法**，没有使用存储过程下创建对象生成 SQL 语句如下：

```
UPDATE Customer SET Version = @p0, Firstname = @p1, Last name = @p2  
WHERE CustomerId = @p3 AND Version = @p4;  
@p0 = '2', @p1 = 'YJingCnBlogs', @p2 = 'Lee', @p3 = '13', @p4 = '1'
```

Step2: **修改映射文件添加存储过程**，打开 Customer.hbm.xml 映射文件，在 Class 元素下添加<sql-update>节点，调用 CustomerUpdate 存储过程，CustomerUpdate 存储过程有四个参数，这里用四个问号表示：

```
<sql-update>exec CustomerUpdate ?,?,?,?</sql-update>
```

Step3: 执行 UpdateCustomerTest()测试方法，出现错误“Row was updated or deleted by another transaction (or unsaved-value mapping was incorrect): [DomainModel.Entities.Customer#15]”，这个错误同删除对象存储过程一样，我们修改 CustomerUpdate 存储过程，去掉 SET NOCOUNT ON，再次执行 UpdateCustomerTest() 测试方法，输出结果如下：

```
exec CustomerUpdate @p0,@p1,@p2,@p3;  
@p0 = '2', @p1 = 'YJingCnBlogs', @p2 = 'Lee', @p3 = '15', @  
p4 = '1'
```

这段根据我们写的存储过程实质 SQL 语句为：

```
UPDATE [Customer] SET [Version] = '2', [Firstname] = 'YJingC  
nBlogs',  
[Lastname] = 'Lee' WHERE [CustomerId] = '1'
```

虽然 NHibernate 知道 Version 列是版本控制，它自动由原来的 1 更新为 2，但是看看上面生成的 SQL 语句还是不怎么舒服，其 @p4 参数无缘无故的在那里。

Step4: 修改 CustomerUpdate 存储过程，把版本控制用好，编写如下：

```
ALTER PROCEDURE [dbo].[CustomerUpdate]  
(  
    @Version int,  
    @Firstname nvarchar(50) = NULL,  
    @Lastname nvarchar(50) = NULL,  
    @CustomerId int,  
    @OldVersion int  
)  
AS  
    UPDATE [Customer]  
    SET  
        [Version] = @Version,  
        [Firstname] = @Firstname,  
        [Lastname] = @Lastname  
    WHERE  
        [CustomerId] = @CustomerId and [Version] = @OldVersion  
    RETURN @@Error
```

Step4: 执行 UpdateCustomerTest()测试方法, 出现错误“过程或函数 'CustomerUpdate' 需要参数 '@OldVersion', 但未提供该参数”, Oh! 映射文件中调用存储过程忘了增加一个参数, 现在是五个参数了!

Step5: **修改存储过程参数数量**, 打开映射文件在<sql-update>中添加一个参数即添加“,?”

```
<sql-update>exec CustomerUpdate ?,?,?,?,?</sql-update>
```

Step6: 执行 UpdateCustomerTest()测试方法, NHibernate 生成语句如下, 这下完美了。

```
exec CustomerUpdate @p0,@p1,@p2,@p3,@p4;  
@p0 = '4', @p1 = 'YJingCnBlogsCnBlogs', @p2 = 'Lee', @p3 =  
'13', @p4 = '3'
```

另外, 如果你不喜欢存储过程的话, 你也可以这样写, 效果和使用存储过程一样。

```
<sql-update>UPDATE [Customer] SET [Version] = ?,[Firstnam  
e] = ?,[Lastname] = ?  
WHERE [CustomerId] =? and [Version] =?</  
sql-update>
```

结语

这一篇和上一篇介绍了如何使用存储过程删除对象、创建对象、更新对象, 还有一种使用<sql-query>来调用存储过程, 非常方便, 下篇继续介绍。

NHibernate 之旅(17): 探索 NHibernate 中使用存储过程(下)

本节内容

- 引入
- 实例分析
- 拾遗
- 结语

引入

上两篇，介绍使用 MyGeneration 提供的模板创建存储过程和删除对象、创建对象、更新对象整个详细过程，这篇介绍如何利用<sql-query>做更多的事，在程序开发中，我们不仅仅只利用存储过程增删查改对象，我们还可以想执行任意的存储过程，这不局限于某个对象，某个 CURD 操作，怎么做呢？注意：本篇并非官方权威的资料，所以敬请参考。如果你还没有学习 NHibernate，请快速链接到 NHibernate 之旅系列文章导航。

实例分析

下面我用几个例子来分析使用<sql-query>来执行存储过程。

1.返回标量

Step1: 存储过程

```
CREATE PROCEDURE scalarSProcs
    @number int
AS
BEGIN
    SELECT @number as value, 'YJingLee' as name
END
```

这里模拟验证键/值对，按键查询名称。这里返回 YJingLee。

Step2: 映射文件

在映射文件中使用<sql-query>并定义<sql-query>查询的名称,使用<return-scalar>元素来指定返回的标量值,并指定字段的别名和类型。调用存储过程时,需要一个参数,这里用命名参数表示,这里打开 Customer.hbm.xml 在 Class 元素上编写如下代码:

```
<sql-query name="ScalarSProcs" >
  <return-scalar column="value" type="int"/>
  <return-scalar column="name" type="string"/>
  exec scalarSProcs :number
</sql-query>
```

Step3: 数据访问方法

在数据访问层中,使用 ISession 接口提供的 GetNamedQuery 方法来调用带命名的存储过程,并传递一个整形参数。代码如下:

```
public IList ScalarStoredProcedure()
{
    return _session.GetNamedQuery("ScalarSProcs")
        .SetInt32("number", 22).List();
}
```

Step4: 测试方法

测试上面的方法,验证其标量返回的结果是否与预期的一致。

```
[Test]
public void ScalarStoredProcedureTest()
{
    IList list = _sprocs.ScalarStoredProcedure();
    object[] o = (object[])list[0];
    Assert.AreEqual(o[0], 22);
    Assert.AreEqual(o[1], "YJingLee");
}
```

OK, 测试成功, NHibernate 生成 SQL 语句如下:

```
exec scalarSProcs @p0; @p0 = '22'
```

2. 设置参数

Step1: 存储过程

```
CREATE PROCEDURE paramSProcs
    @i int,
```



```

    @j int
AS
BEGIN
    SELECT @i as value, @j as value2
END

```

这里模拟一个存储过程，学习如何运用参数。

Step2: 映射文件

带参数的存储过程，参数有两种写法，一种是使用“?”参数来表示：

```

<sql-query name="ParamSProcs">
    <return-scalar column="value" type="long"/>
    <return-scalar column="value2" type="long"/>
    exec paramSProcs ?, ?
</sql-query>

```

另外一种混合方式，使用“?”参数和命名参数表示：

```

<sql-query name="ParamSProcs">
    <return-scalar column="value" type="long" />
    <return-scalar column="value2" type="long" />
    exec paramSProcs ?, :second
</sql-query>

```

Step3: 数据访问方法

在数据访问层中，我们同样使用 `GetNamedQuery` 方法来调用命名的 `<sql-query>` 存储过程，需要传递两个参数，这里支持位置参数和命名参数，其方法如下所示：

```

public IList ParamStoredProcedure()
{
    return _session.GetNamedQuery("ParamSProcs")
        .SetInt64(0, 10L)
        .SetInt64(1, 20L)
        //或者.SetInt64("second", 20L)
        .List();
}

```

Step4: 测试方法

编写一个测试用例测试上面的方法，OK！

```
[Test]
```

```

public void ParamStoredProcedureTest()
{
    IList list = _sprocs.ParamStoredProcedure();
    object[] o = (object[])list[0];
    Assert.AreEqual(o[0], 10L);
    Assert.AreEqual(o[1], 20L);
}

```

3. 实体查询

Step1: 存储过程

上面的查询都是返回标量值的，也就是返回的是“裸”数据。我需要查询 Customer 实体对象，怎么办呢？编写一个存储过程按 CustomerId 查询 Customer。

```

CREATE PROCEDURE [entitySProcs]
    @CustomerId int
AS
BEGIN
    SELECT * FROM [Customer]
    WHere [CustomerId] =@CustomerId
END

```

Step2: 映射文件

通过命名查询来映射这个存储过程，使用 return 返回具体的实体类，使用 <return-property> 告诉 NHibernate 使用哪些属性值，允许我们来选择如何引用字段以及属性。

```

<sql-query name="EntitySProcs">
    <return class="DomainModel.Entities.Customer,DomainModel" />
    <return-property name="CustomerId" column="CustomerId" />
    <return-property name="Version" column="Version" />
    <return-property name="Firstname" column="Firstname" />
    <return-property name="Lastname" column="Lastname" />
    </return>
    exec entitySProcs :customerId
</sql-query>

```

这非常奇怪，我使用这种方式测试不通过，晚上和 [Gray Zhang](#) 讨论这个问题，[Gray Zhang](#) 使用这个方式运行成功，在我的机器上运行不成功，始终出现“NHibernate.ADOException : could not execute query [exec entitySProcs ?] Name:customerId - Value:1 [SQL: exec entitySProcs ?]”问题，我换了一下方式，去掉 <return-property> 元素，像这样：

```
<sql-query name="EntitySProcs">
  <return class="DomainModel.Entities.Customer,DomainModel"/>
  exec entitySProcs :customerId
</sql-query>
```

Step3: 数据访问方法

```
public Customer EntityStoredProcedure(int customerId)
{
    return _session.GetNamedQuery("EntitySProcs")
        .SetInt32("customerId",customerId)
        .UniqueResult<Customer>();
}
```

Step4: 测试方法

```
[Test]
public void EntityStoredProcedureTest()
{
    Customer customer = _sprocs.EntityStoredProcedure(1);
    int customerId = customer.CustomerId;
    Assert.AreEqual(1, customerId);
}
```

Step5: 返回实体属性

有个需求，我不想返回整个实体 Customer，我想返回实体的 Firstname 属性怎么办呢？哦，反应过来了，使用返回标量的方法，我来写下 <sql-query>：

```
<sql-query name="SingleEntitySProcs">
  <return-scalar column="Firstname" type="string" />
  exec singleEntitySProcs
</sql-query>
```

注意存储过程当前仅仅返回标量和实体。接下来就留给大家完成吧！给个提示，先写存储过程，然后写访问方法，最后测试一下看看是否返回的 Firstname 列表了。

拾遗

使用存储过程查询无法使用 `setFirstResult()/setMaxResults()` 进行分页。

存储过程的参数的位置顺序是非常重要的，必须和 **NHibernate** 持久化类属性顺序相同。

你可以在存储过程里设定 `SET NOCOUNT ON`，这可能会提高效率。

我们可以在类映射里使用 `<loader query-ref="EntitySProcs"/>` 引用这个命名查询定制装载存储过程。

结语

好了，通过三篇文章的介绍，知道了如何在 **NHibernate** 使用存储过程来删除对象、创建对象、更新对象、查询对象等操作，还有一些零碎的东西就在于大家在平时学习中去探索了。**NHibernate** 之旅系列中关于存储过程的内容就说到这里吧，下篇开始看看如何使用代码生成器。

NHibernate 之旅(18): 初探代码生成工具使用

本节内容

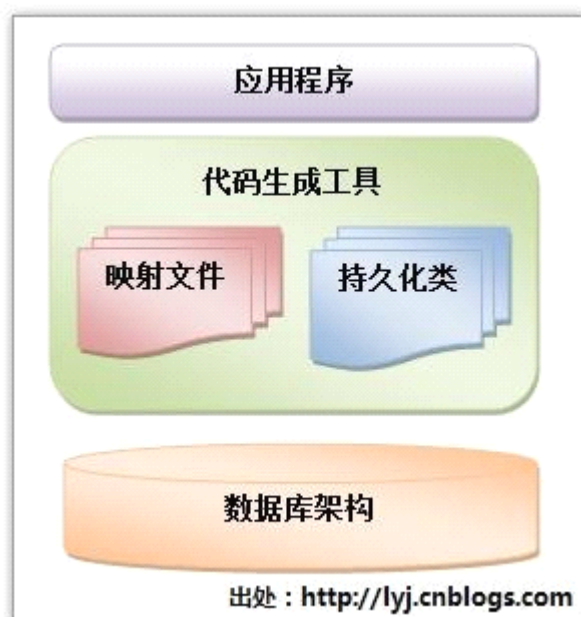
- 引入
- 代码生成工具
- 结语

引入

我们花了大量的篇幅介绍了相关 NHibernate 的知识，一直都是带着大家手动编写代码，首先创建数据库架构，然后编写持久化类和映射文件，最后编写数据操作方法，测试方法。这是典型的数据库驱动开发(DbDD, Database-Driven Development)技术，但是自己不是这样做的，我先编写持久化类和映射文件，然后偷偷的使用 SchemaExport 工具把数据库生成了，按上面的步骤写文章的，关于 SchemaExport 工具就是下一篇文章的事情了，这篇说说利用数据库架构用代码生成工具生成持久化类和映射文件。

所谓数据库驱动模型是指对象模型随着数据库架构改变而改变，那么我们为什么还使用这个技术呢？有两种原因：你的数据库是遗留下来的系统使用的，数据库中已经存在大量有用数据，不可以更换数据库就将就着使用了；你的数据库架构按照需求分析基本上确定了，不要做任何改动，在系统设计初期已经把数据库建好了。

让我们用一张图片来大致展示典型的数据库驱动开发模型。以数据库架构为核心。



使用数据库驱动模型流程就是当数据库架构修改时候，映射文件和持久化类通过代码生成工具重新生成一下。我们只要修改应用程序即可。

这里延伸一个问题：数据库是否需要在项目开始前设计？

如果我们进行数据库设计，那么就产生一系列问题：我们在面向对象领域设计持久化对象必须考虑事先设计好的数据库表结构以及表关系，在编写映射文件时候也要考虑，在面向对象中的继承、多态等特性根本没法使用。所以我不推荐在项目开始设计数据库，大家认为如何？所以使用代码生成工具是下下策。

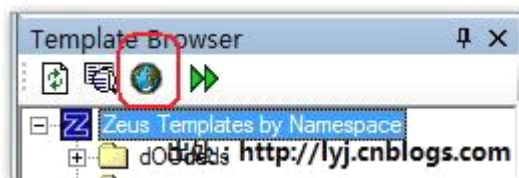
代码生成工具

顾名思义，代码生成工具大家都使用过，有商业的 [CodeSmith](#) 开源的 [MyGeneration](#) 等，这篇我们介绍如何使用开源 [MyGeneration](#) 代码生成工具根据数据库架构生成映射文件和持久化类。如果你机器上还没有安装 [MyGeneration](#)，请到[这里](#)下载，然后安装 [MyGeneration](#)，安装之后，打开 [MyGeneration](#)，配置“默认设置”、打开“模板浏览器”窗口，这些步骤如果不清楚的话请转向 [NHibernate 之旅\(15\)：探索 NHibernate 中使用存储过程\(上\)](#)文章中吧，具体介绍了上面的步骤和方法。

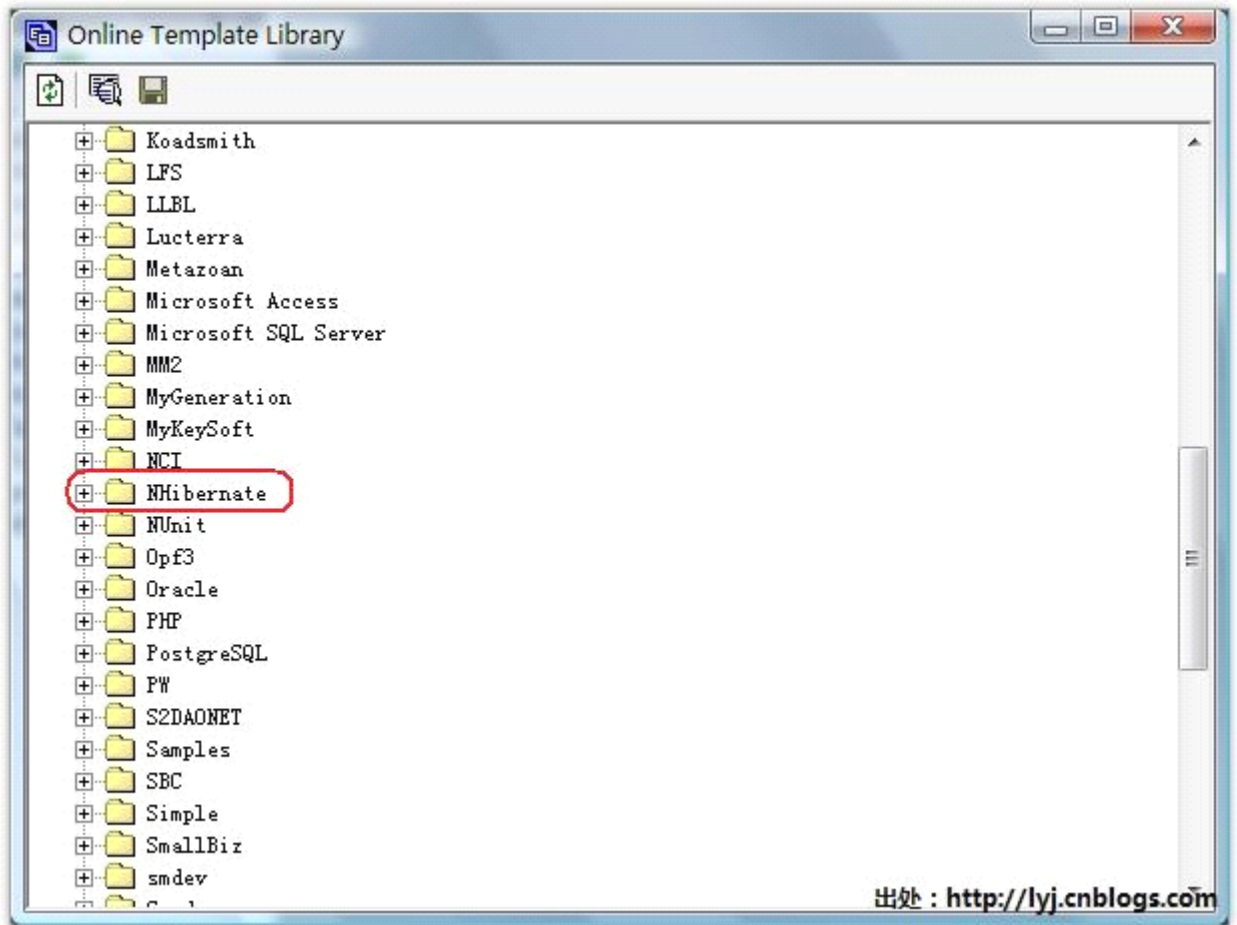
再提示一下：如果你的操作系统是 Windows Vista，请右击“以管理员身份运行”[MyGeneration](#)。

这篇我们接着做：

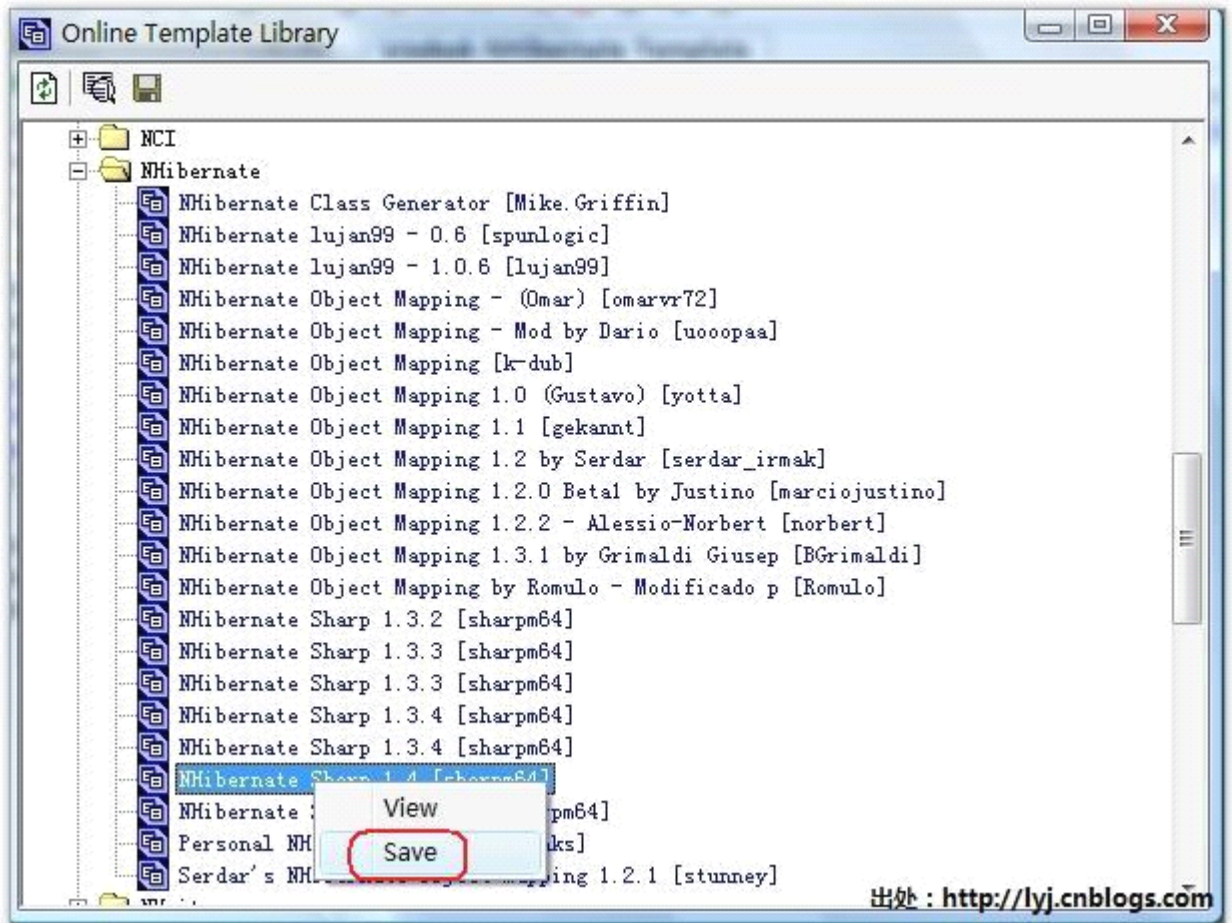
Step1: 点击“模板浏览器”窗口的第三个“在线更新”按钮：来在线下载模板。



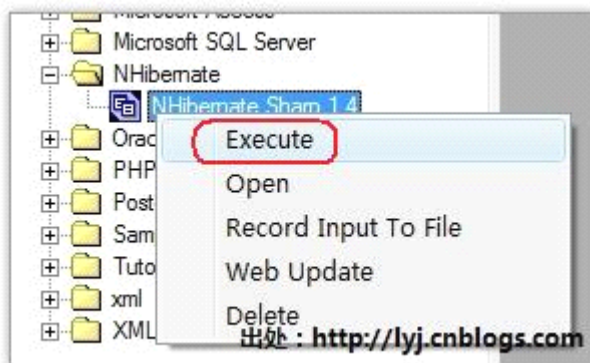
Step2: 出现“在线模板库”窗口，在“在线模板库”中提供了各种各样的模板，找到 [NHibernate](#) 节点：



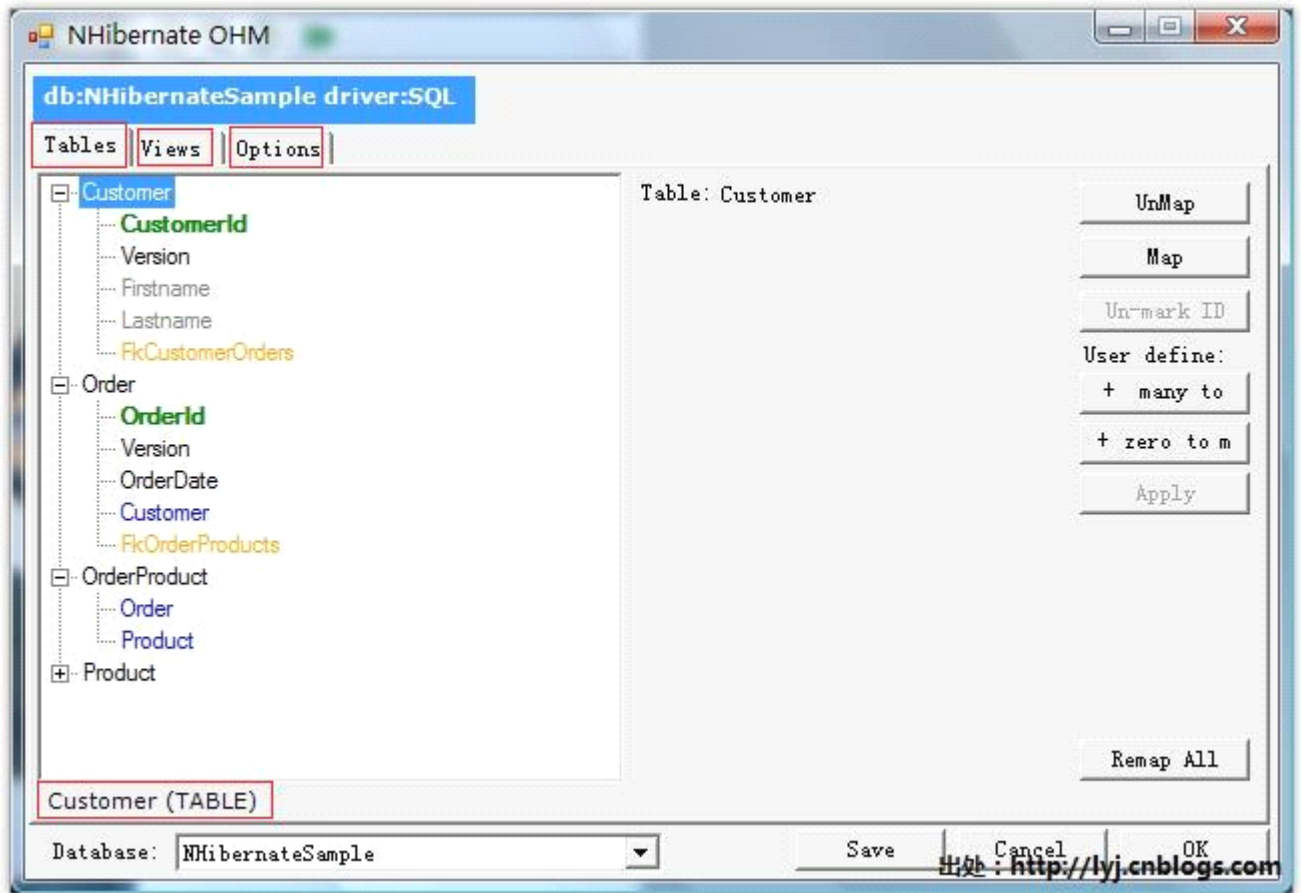
Step3: 展开 NHibernate 节点，找到“NHibernate Sharp 1.4 [sharpm64]”模板，右键点击“保存”。这个模板就自动保存到本地模板文件夹中。



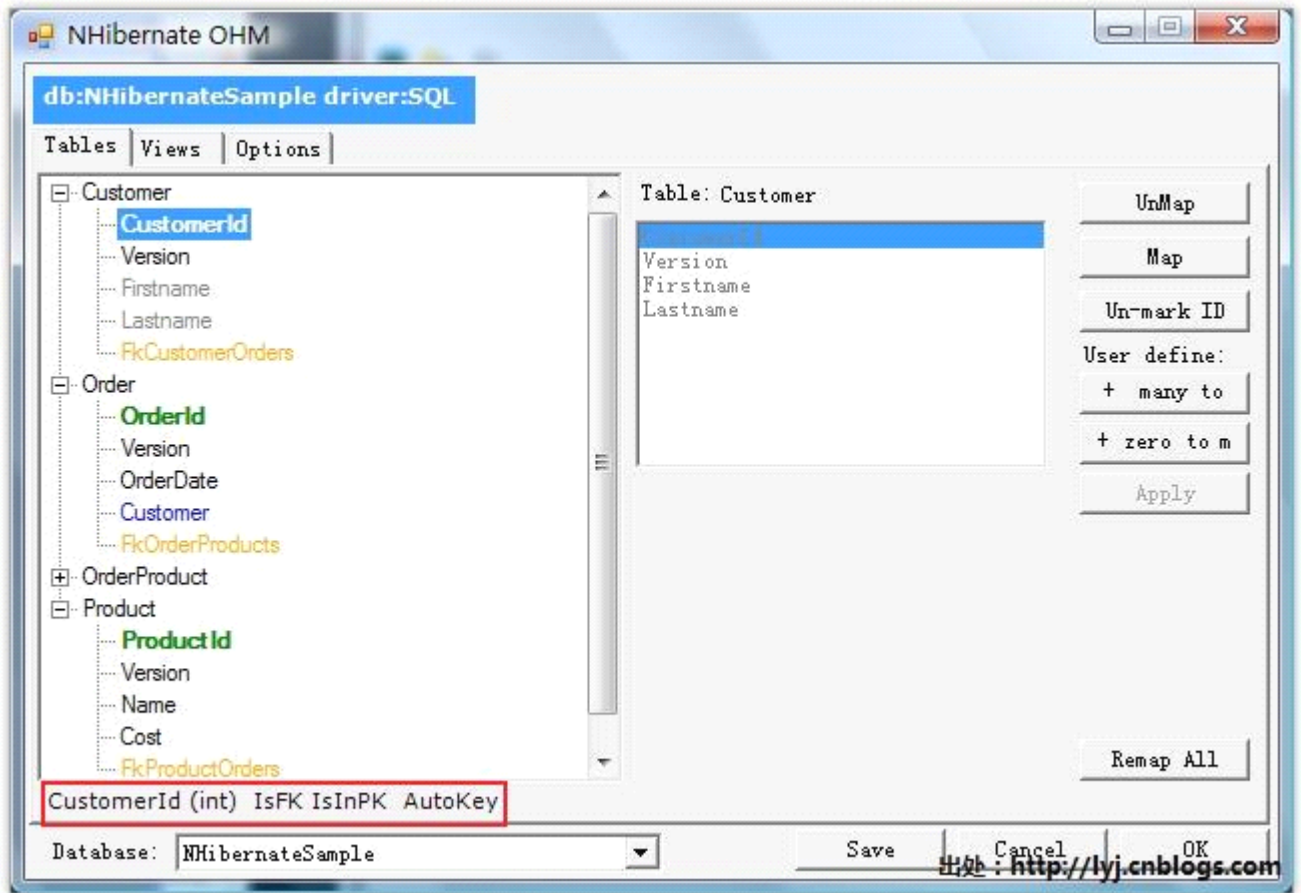
Step4: 点击“模板浏览器”窗口的第一个“刷新”按钮，这个模板就在“模板浏览器”可以看见了，展开 NHibernate 节点，右击“执行”NHibernate Sharp1.4 模板。



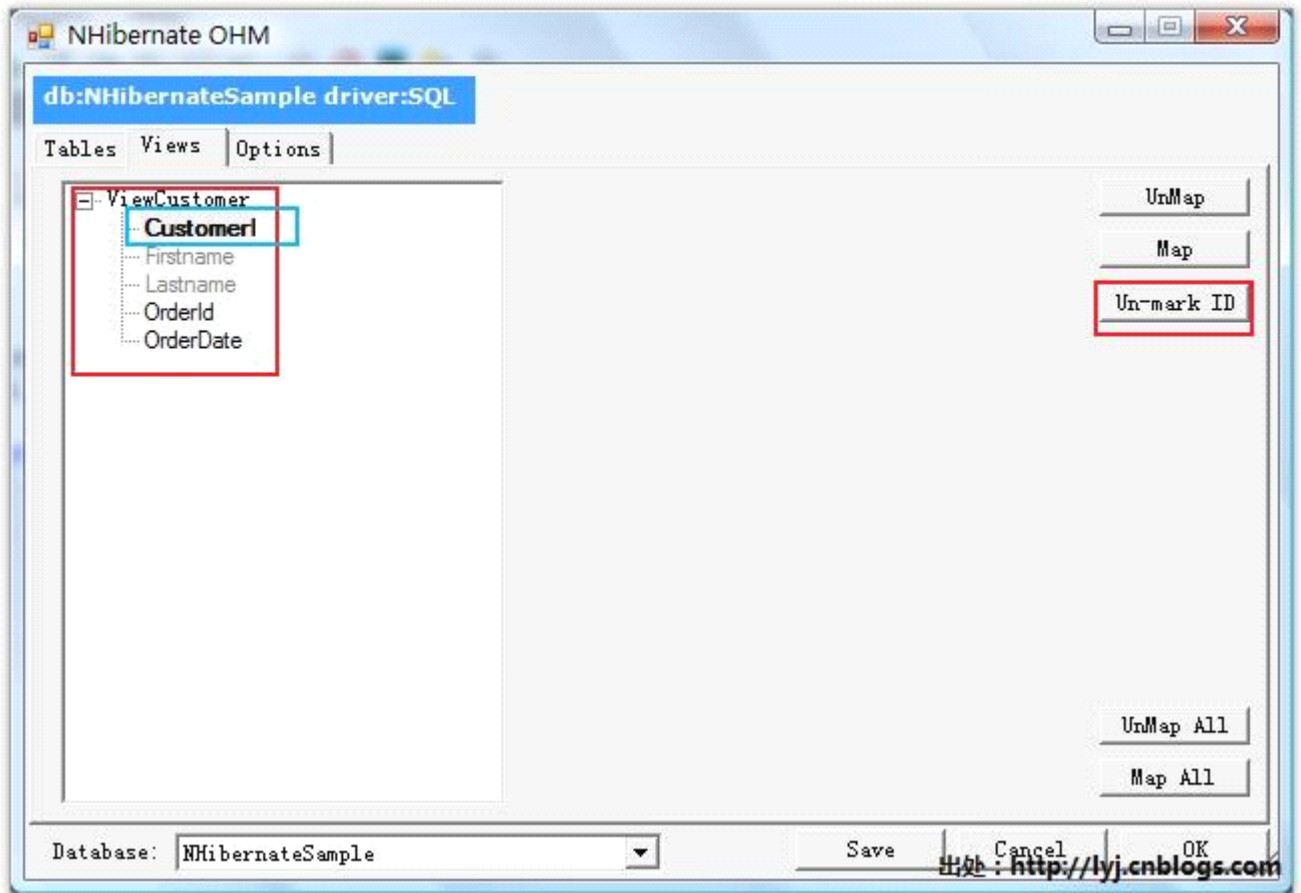
Step5: 这就是 NHibernate OHM 界面窗口，右面显示表、视图、操作标签，在表标签界面上，右边有不映射、映射等按钮，在表中不同的颜色代表字段不同的属性。



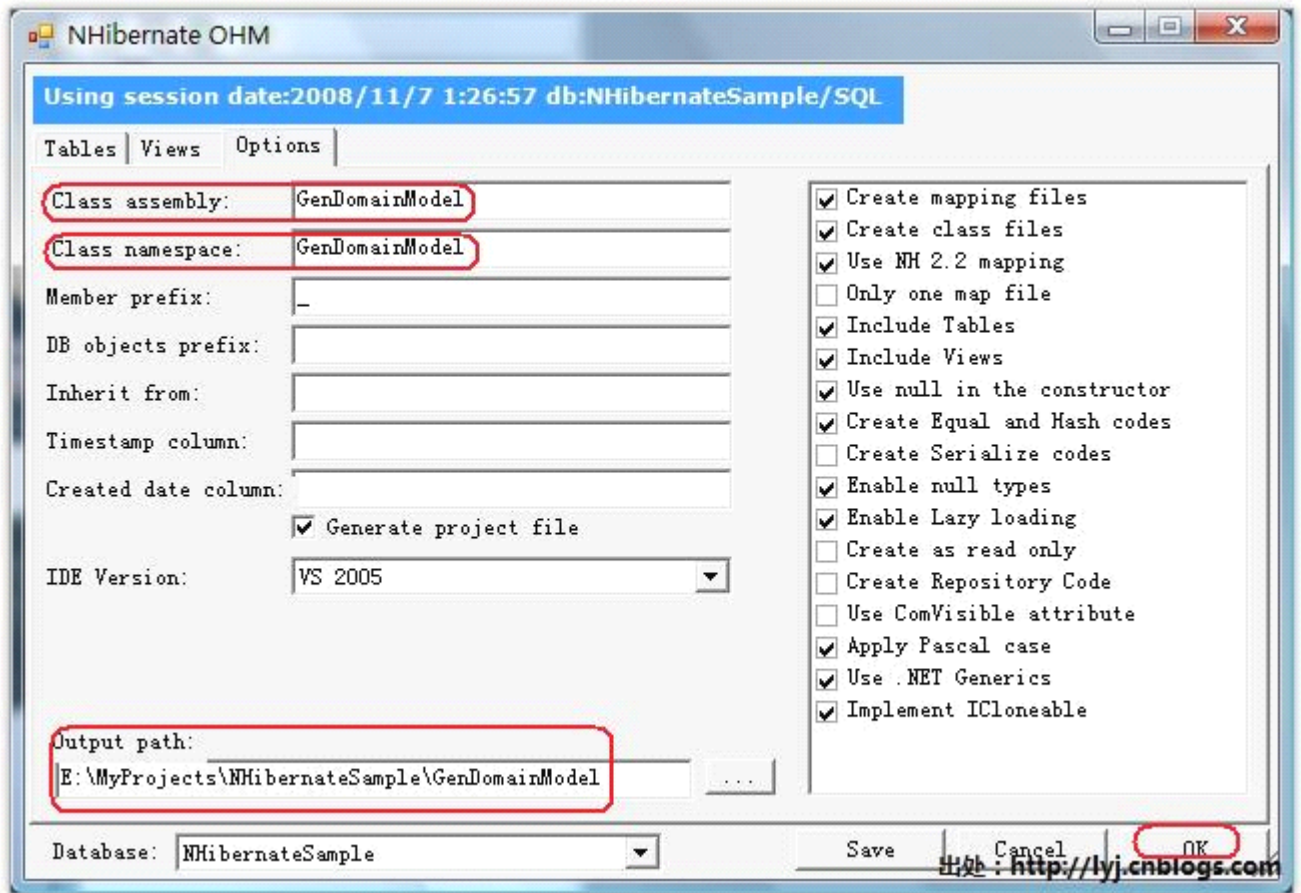
Step6: 点击“CustomerId”列，右边显示了这个表，并在状态栏显示了“CustomerId”列的属性：int 类型、主键。另外可以自己摸索一下：



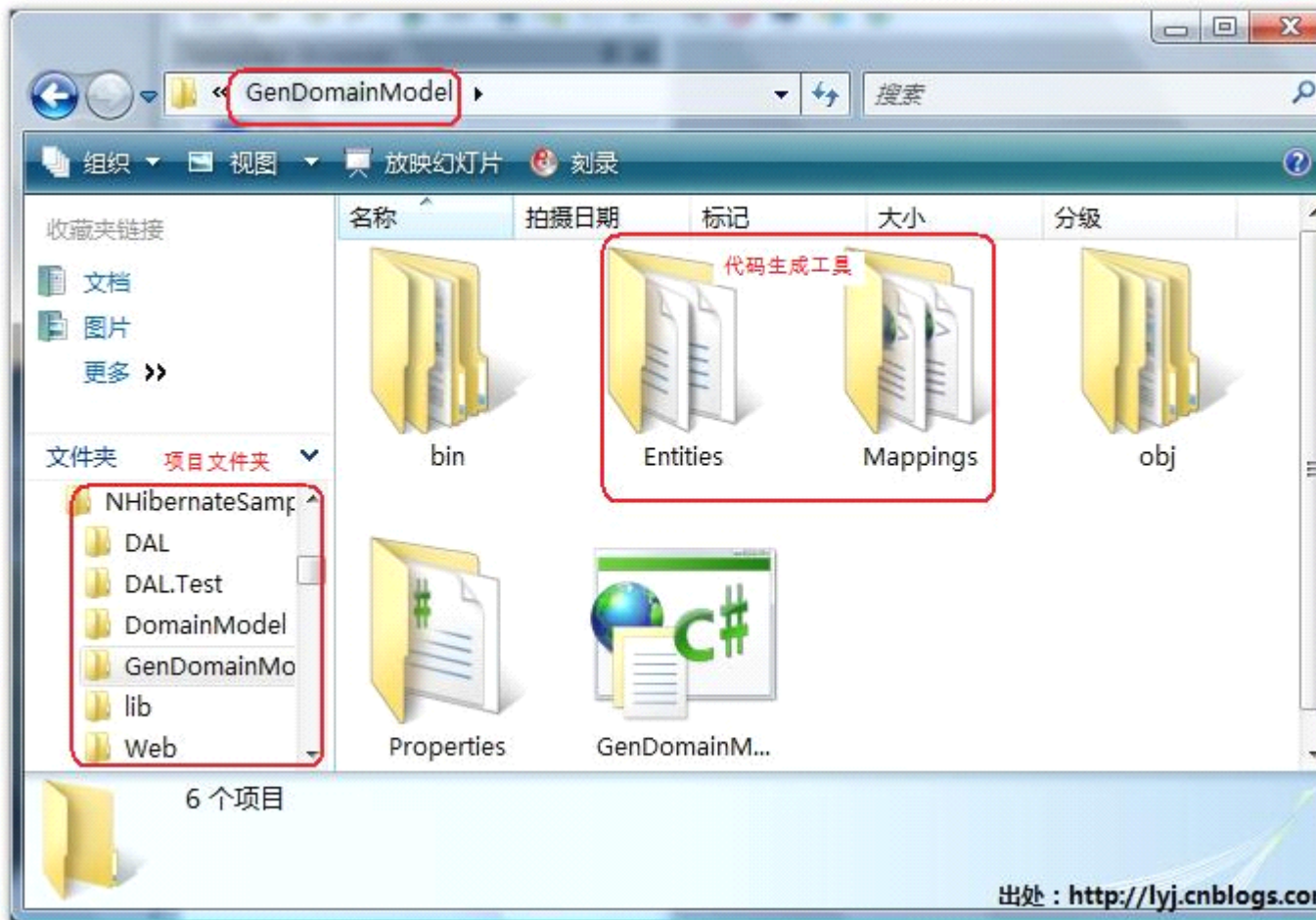
Step7: 点击“视图”标签，设置 CustomerId 为主键。



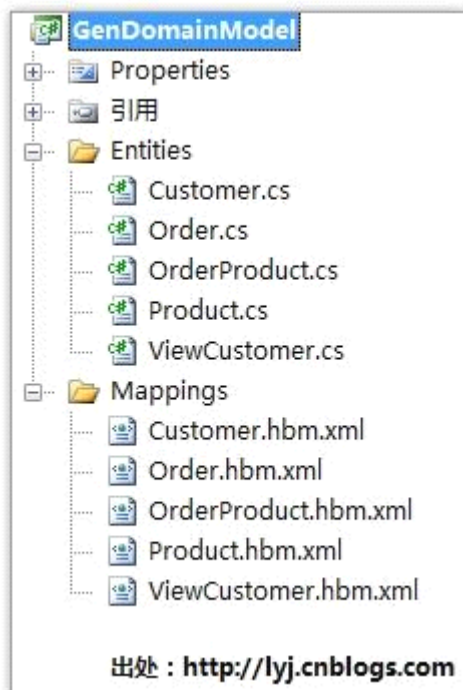
Step8: 点击“操作”标签，我们具体设置程序集名称和命名空间名称，这里我设置 GenDomainModel，设置生成工程，由于这个模板不支持 VS2008，所以选择生成 IDE 版本为 VS2005。设置输出路径。点击 OK 按钮。



Step9: 这时打开文件夹，代码生成工具根据数据库架构生成了相应的持久化类和映射文件。



Step10: 用 VS2008 打开, VS2008 自动升级 GenDomainModel.csproj 解决方案文件, 解决方案项目文件如下:



Step11: 这时就可以使用了，可以把这个类库利用添加现有项目功能添加到我们的项目中作为实体持久层，我们利用这个实体持久层编写数据访问层方法实现对数据库的 CRUD 操作，当然在使用前要搞清楚这个项目的类库架构，另外由于模板本身还不是很完善（例如这个模板不支持版本控制映射、不支持多对多直接映射），所以我们还需要按照实际情况去修改持久化类和映射，显然无形中增加了一些负担。

注意不要忘记在 `hibernate.cfg.xml` 中修改 `<mapping assembly="DomainModel"/>` 为 `<mapping assembly="GenDomainModel"/>`。

结语

本身对于代码生成工具的使用很简单的，就是下一步下一步的按，这节就是多图展示一下怎么快速使用代码生成工具，在 `MyGeneration` 中还有很多模板用于生成 `NHibernate` 的持久化类和映射文件，每个模板都有自己的缺点和优点，生成不同的结构，这在于大家去发现了~~但是我发现还是自己手写代码才是霸道！思路清晰，代码简单！

至今不明白这个问题：代码生成工具到底给谁用的？新手？老鸟？唯一的好处就是快？不用写代码。但是你知道它生成的架构吗？它的思路吗？

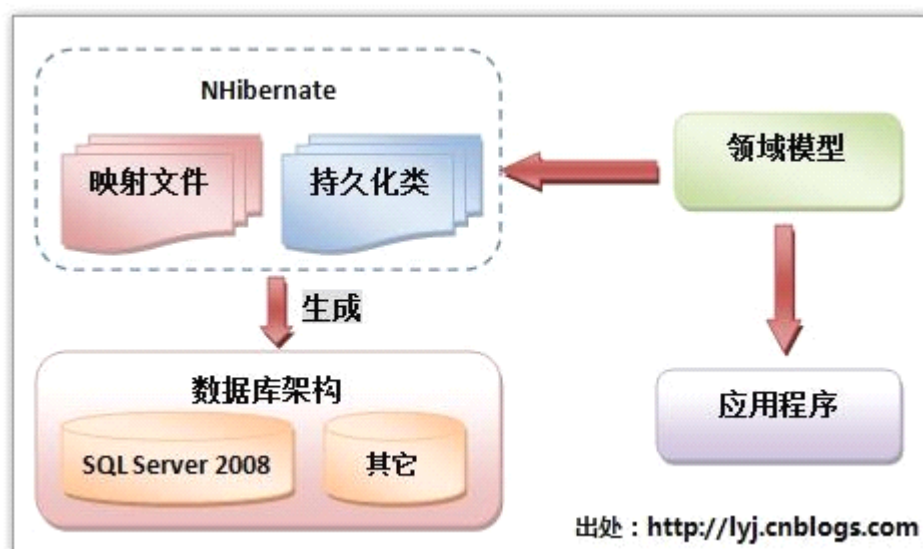
NHibernate 之旅(19): 初探 SchemaExport 工具使用

本节内容

- 引入
- SchemaExport 工具
- SchemaUpdate 工具
- 实例分析
- 结语

引入

我其实都是一直先编写持久化类和映射文件，然后使用 SchemaExport 工具生成数据库架构。这样的方式就是领域驱动设计/开发(DDD, Domain Driven Design/Development)。我的理解是系统的设计应该基于对象模型，主要考虑对象的设计和逻辑上，然后按照对象模型建立数据库关系模型，这才是现在面向对象开发的步骤，并不是上一篇先设计数据库然后再设计对象。用一幅图可以形象的说明领域驱动设计：



当在设计时，我们的领域模型需要改变，只需修改 NHibernate 结构和应用程序，不需要修改数据库架构，只要利用 SchemaExport 工具重新生成数据库架构就可以了。但是使用数据库只是其中一种方式，我们也可以使用 XML 文件来存储数据。

SchemaExport 工具

NHibernate 的 hbm2dll 提供 SchemaExport 工具：给定一个连接字符串和映射文件，不需输入其他东西就可以按照持久化类和映射文件自动生成数据库架构，现在 SchemaExport 工具还不是很强大，但是一般应用足够了，它还是一个相当原始的 API 还在不断改进。

SchemaExport 工具就是把 DDL 脚本输出到标准输出，同时/或者执行 DDL 语句。SchemaExport 工具提供了三个方法，分别是 Drop()、Create()、Execute()，前两个方法实质是调用 Execute()方法。通常使用 Execute()方法来生成数据库架构的。

SchemaUpdate 工具

在 NHibernate2.0 中新添加 SchemaUpdate 工具，可以用来更新数据库架构。但是我觉得没有什么作用，因为它不能 Drop 现有的表或列，也不能更新现有的列，只能添加新的表和列。如果我需要删除表或者列或者修改其中列，SchemaUpdate 工具就显得无能为力了。

实例分析

知道了上面的知识就好好实战一下：看看具体怎么使用呢？

1.SchemaExport 工具实战

通常我们使用生成数据库架构代码实例像这样：

```
Configuration cfg=new Configuration();
cfg.Configure();
SchemaExport export =new SchemaExport (cfg);
export.Execute(...);
```

1.准备工作

现在数据访问测试层新建一 SchemaExportFixture.cs 文件用于测试生成实战。声明一个全局变量 _cfg，编写 [SetUp]方法在每个测试方法执行之前调用：

```
[TestFixture]
public class SchemaExportFixture
{
    private Configuration _cfg;
    [SetUp]
    public void SetupContext()
    {
```



```
        _cfg = new Configuration();
        _cfg.Configure();
    }
    //测试.....
}
```

2.测试 Drop(script, export)方法

```
[Test]
public void DropTest()
{
    var export = new SchemaExport(_cfg);
    export.Drop(true, true);
}
```

Drop(script, export)方法根据持久类和映射文件执行删除数据库架构。有两个参数，第一个为 True 就是把 DDL 语句输出到控制台，第二个为 True 就是根据持久类和映射文件执行删除数据库架构操作，经过调试可以发现 Drop(script, export)方法其实是执行了 Execute(script, export, true, true)方法。

3.测试 Create(script, export)方法

```
[Test]
public void CreateTest()
{
    var export = new SchemaExport(_cfg);
    export.Create(true, true);
}
```

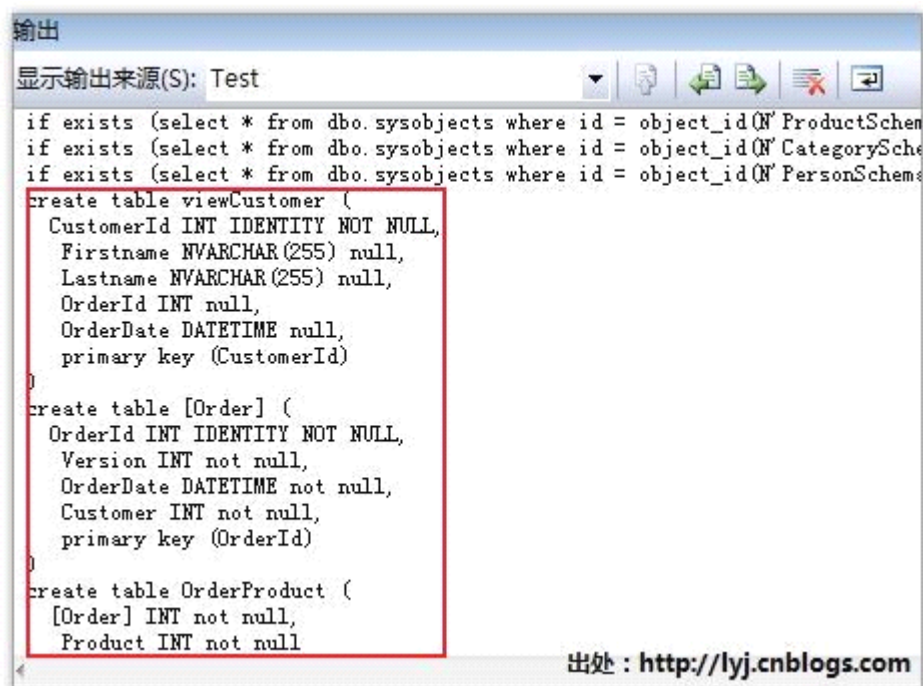
Create(script,export)方法根据持久类和映射文件先删除架构后创建删除数据库架构。有两个参数，第一个为 True 就是把 DDL 语句输出到控制台，第二个为 True 就是根据持久类和映射文件先执行删除再执行创建操作，经过调试可以发现这个方法其实是执行 Execute(script,export, false, true)方法。

4.测试 Execute(script, export, justDrop, format)方法

```
[Test]
public void ExecuteTest()
{
    var export = new SchemaExport(_cfg);
    export.Execute(true, true, false, false);
}
```

Execute(script, export, justDrop, format)方法根据持久类和映射文件先删除架构后创建删除数据库架构。有四个参数，第一个为 True 就是把 DDL 语句输出到控制台；第二个为 True 就是根据持久类和映射文件在数据库中先执行删除再执行创建操作；第三个为 false 表示不是仅仅执行 Drop 语句还执行创建操作，这个参数的不同就扩展了上面两个方法；第四个参数为 false 表示不是格式化输出 DDL 语句到控制台，是在一行输出的。

所谓格式化输出就像这样：



The screenshot shows a window titled "输出" (Output) with a toolbar and a text area containing SQL code. The code is formatted with line breaks and indentation. A red box highlights the first two table creation statements. The text area also includes a URL at the bottom right: "出处: http://lyj.cnblogs.com".

```
if exists (select * from dbo.sysobjects where id = object_id(N' ProductSchem
if exists (select * from dbo.sysobjects where id = object_id(N' CategorySche
if exists (select * from dbo.sysobjects where id = object_id(N' PersonSchem
create table viewCustomer (
  CustomerId INT IDENTITY NOT NULL,
  Firstname NVARCHAR(255) null,
  Lastname NVARCHAR(255) null,
  OrderId INT null,
  OrderDate DATETIME null,
  primary key (CustomerId)
)
create table [Order] (
  OrderId INT IDENTITY NOT NULL,
  Version INT not null,
  OrderDate DATETIME not null,
  Customer INT not null,
  primary key (OrderId)
)
create table OrderProduct (
  [Order] INT not null,
  Product INT not null
```

出处: <http://lyj.cnblogs.com>

一行输出就像这样：

```
if exists (select * from dbo.sysobjects where id = object_id(N' [Order]') and OBJECTPROPERTY(
if exists (select * from dbo.sysobjects where id = object_id(N' OrderProduct') and OBJECTPROF
if exists (select * from dbo.sysobjects where id = object_id(N' Product') and OBJECTPROPERTY(
if exists (select * from dbo.sysobjects where id = object_id(N' Customer') and OBJECTPROPERTY
if exists (select * from dbo.sysobjects where id = object_id(N' ProductSchema') and OBJECTPRO
if exists (select * from dbo.sysobjects where id = object_id(N' CategorySchema') and OBJECTPF
if exists (select * from dbo.sysobjects where id = object_id(N' PersonSchema') and OBJECTPROF
create table viewCustomer (CustomerId INT IDENTITY NOT NULL, Firstname NVARCHAR(255) null,
create table [Order] (OrderId INT IDENTITY NOT NULL, Version INT not null, OrderDate DATETIME
create table OrderProduct ([Order] INT not null, Product INT not null)
create table Product (ProductId INT IDENTITY NOT NULL, Version INT not null, Name NVARCHAR(50)
create table Customer (CustomerId INT IDENTITY NOT NULL, Version INT not null, Firstname NV
create table ProductSchema (Id UNIQUEIDENTIFIER not null, Name NVARCHAR(50) not null unique,
create table CategorySchema (Id UNIQUEIDENTIFIER not null, Name NVARCHAR(50) not null, prim
create table PersonSchema (Id UNIQUEIDENTIFIER not null, FirstName NVARCHAR(50) not null, L
alter table [Order] add constraint FK_CustomerOrders foreign key (Customer) references Custo
alter table OrderProduct add constraint FK_ProductOrders foreign key (Product) references Pr
alter table OrderProduct add constraint FK_OrderProducts foreign key ([Order]) references [O
create index IDX_Product_Name on ProductSchema (Name)
alter table ProductSchema add constraint FK_Product_Category foreign key (CategorySchema) re
drop table dbo.viewCustomer
```

出处 : <http://lyj.cnblogs.com>

5.测试 Execute(script, export, justDrop, format, connection, exportOutput)方法

```
[Test]
public void ExecuteOutTest()
{
    var export = new SchemaExport( cfq);
    var sb = new StringBuilder();
    TextWriter output = new StringWriter(sb);
    export.Execute(true, false, false, false, null, output);
}
```

Execute(script, export, justDrop, format, connection, exportOutput)方法根据持久类和映射文件先删除架构后创建删除数据库架构。有六个参数，第一个为 True 就是把 DDL 语句输出到控制台；第二个为 false 就是不执行 DDL 语句；第五个为自定义连接。当 export 为 true 执行语句时必须打开连接。该方法不关闭连接，null 就是使用默认连接，最后一个参数自定义输出，这里我输出到 TextWriter 中。

2.SchemaUpdate 工具实战

现在数据访问测试层新建一 SchemaUpdateFixture.cs 文件用于测试生成实战。先声明一个全局变量_cfg:

```
private Configuration _cfg;
```

这里我用另外一种方式配置映射文件，先定义两个映射 XML 分别代表旧的和新的这样才能体现测试更新数据库架构的意义。

旧映射 XML：这里我使用 **Product** 持久化类，由于在之前我们定义了 **Product** 持久化类，这里直接模拟定义映射 XML：拥有主键 **ProductId** 和 **Name** 字段。

```
public const string product_xml =
    "<?xml version='1.0' encoding='utf-8' ?>" +
    "<hibernate-mapping xmlns='urn:nhibernate-mapping-2.2"
    "' +
    "    assembly='DomainModel'" +
    "    namespace='DomainModel'" +
    "    <class name='DomainModel.Entities.Product,DomainMo"
    "del'" +
    "        <id name='ProductId'" +
    "            <generator class='native'/" +
    "        </id'" +
    "        <property name='Name'/" +
    "    </class'" +
    "</hibernate-mapping>";
```

新映射 XML：更新上面映射 XML：主键 **ProductId**(没有改变)；**Name** 字段：添加不可为空和长度为 50；另外增加了 **Cost** 字段，类型为 **float** 不可为空。

```
public const string newproduct_xml =
    "<?xml version='1.0' encoding='utf-8' ?>" +
    "<hibernate-mapping xmlns='urn:nhibernate-mapping-2.2"
    "' +
    "    assembly='DomainModel'" +
    "    namespace='DomainModel'" +
    "    <class name='DomainModel.Entities.Product,DomainMo"
    "del'" +
    "        <id name='ProductId'" +
    "            <generator class='native'/" +
    "        </id'" +
    "        <property name='Name' not-null='true' length='50'"
    "/>" +
    "        <property name='Cost' type='float' not-null='true'/" +
    "    </class'" +
```

```
"</hibernate-mapping>";
```

测试前利用旧映射 XML 创建数据库架构：使用[SetUp]在测试前执行，按照旧映射 XML 创建数据库架构并格式化输出 DDL 语句：

[SetUp]

```
public void SetupContext()
```

```
{
```

```
    //模拟旧系统
```

```
    _cfg = new Configuration();
```

```
    _cfg.Configure();
```

```
    _cfg.AddXml(product_xml);
```

```
    var export = new SchemaExport(_cfg);
```

```
    export.Execute(true, true, false, true);
```

```
}
```

测试更新数据库架构：使用 SchemaUpdate 类提供的唯一的 Execute(script, doUpdate)方法按照新映射 XML 更新数据库架构：

[Test]

```
public void UpdateExistingDatabaseSchemaTest()
```

```
{
```

```
    _cfg = new Configuration();
```

```
    _cfg.Configure().AddXml(newproduct_xml);
```

```
    var update = new SchemaUpdate(_cfg);
```

```
    update.Execute(true, true);
```

```
}
```

测试输出结果如图所示，如果你觉得不放心再看看数据库 Product 表。

```
输出
显示输出来源(S): Test
----- Test started: Assembly: DAL.Test.dll -----
if_exists (select * from dbo.sysobjects where id = object_id(N'Product') and  接下句
create table Product (  接上句 OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table Product
    ProductId INT IDENTITY NOT NULL,
    Name NVARCHAR(255) null,
    primary key (ProductId)
)
alter table Product add Cost REAL

1 passed, 0 failed, 0 skipped, took 4.60 seconds.
出处: http://lyj.cnblogs.com
```

看到了吗？这显然不是我要求的，首先按照旧映射 XML 创建了数据库架构，但是更新数据库架构显得无能为力，仅仅增加了 Cost 字段，我想更新 Name 字段属性为不可为空和长度为 50，但是 SchemaUpdate 工具不能做到！我觉得这个类目前还没有什么作用，期待下一个版本来完善。

结语

这篇文章通过实例介绍 NHibernate 中提供两个实用工具 SchemaExport 工具利用持久化类和映射文件生成数据库架构。SchemaUpdate 工具通过持久化类和映射文件更新数据库架构。

NHibernate 之旅(20): 再探 SchemaExport 工具使用

本节内容

- 引入
- 实例分析
 - 1.表及其约束
 - 2.存储过程、视图
- 结语

引入

上篇我们初步探索了 SchemaExport 工具使用,知道如何使用 SchemaExport 工具和 SchemaUpdate 工具利用 NHibernate 持久化类和映射文件删除、创建、更新数据库架构,这篇具体分析如何为表字段增加一些约束?如何生成存储过程?如何生成视图?使用 SchemaExport 工具帮你搞定。

实例分析

1.表及其约束

众所周知,SchemaExport 工具根据映射文件来生成数据库架构,在映射文件中通过 Class 映射可以很方便的生成数据库表。但是这篇我们看看映射的条件,所以我重新定义两个实体 CategorySchema 和 ProductSchema,一对多关系。

Step1: 两个实体持久化类编写代码如下:

```
public class CategorySchema
{
    public virtual Guid Id { get; set; }
    public virtual string Name { get; set; }
}
public class ProductSchema
{
    public virtual Guid Id { get; set; }
```

```
public virtual string Name { get; set; }
public virtual int UnitsOnStock { get; set; }
public virtual CategorySchema CategorySchema { get; set; }
}
```

Step2: 为两个实体映射，使用最简方式，编写代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="DomainModel" namespace="DomainModel">

    <class name="DomainModel.Entities.CategorySchema,DomainModel">
        <id name="Id">
            <generator class="guid"/>
        </id>
        <property name="Name"/>
    </class>

    <class name="DomainModel.Entities.ProductSchema,DomainModel">
        <id name="Id">
            <generator class="guid"/>
        </id>
        <property name="Name"/>
        <many-to-one name="CategorySchema"
            class="DomainModel.Entities.CategorySchema,DomainModel"/>
    </class>

</hibernate-mapping>
```

Step3: 编写测试用例用于生成数据库架构：

```
[Test]
public void ExecuteSchemaTest()
{
    var export = new SchemaExport(_cfg);
```



```
export.Execute(true, true, false, true);  
}
```

Step4: 测试! NHibernate 生成语句如下:

```
if exists (select 1 from sys.objects  
          where object_id = OBJECT_ID(N'[FK45BBFB51BC95  
15A6]')  
          AND parent_object_id = OBJECT_ID('ProductSchem  
a'))  
alter table ProductSchema drop constraint FK45BBFB51BC95  
15A6  
  
if exists (select * from dbo.sysobjects  
          where id = object_id(N'ProductSchema')  
          and OBJECTPROPERTY(id, N'IsUserTable') = 1)  
drop table ProductSchema  
if exists (select * from dbo.sysobjects  
          where id = object_id(N'CategorySchema')  
          and OBJECTPROPERTY(id, N'IsUserTable') = 1)  
drop table CategorySchema  
  
create table ProductSchema (  
  Id UNIQUEIDENTIFIER not null,  
  Name NVARCHAR(255) null,  
  CategorySchema UNIQUEIDENTIFIER null,  
  primary key (Id)  
)  
create table CategorySchema (  
  Id UNIQUEIDENTIFIER not null,  
  Name NVARCHAR(255) null,  
  primary key (Id)  
)  
  
alter table ProductSchema add constraint FK45BBFB51BC9515  
A6  
foreign key (CategorySchema) references CategorySchema
```

仔细看看生成的语句, 都按默认的值生成了表, Name 列字符串类型 NVARCHAR(255), 默认为 null; 外键默认一数字字符串等。

1. 设置非空类型和长度

在映射文件中为 ProductSchema 实体的 Name 属性添加：not-null="true" 表示非空类型，length="50"：列长度设置为 50，代码片段如下：

```
<property name="Name" not-null="true" length="50"/>
```

测试，生成语句如下：

```
create table ProductSchema (  
    Id UNIQUEIDENTIFIER not null,  
    Name NVARCHAR(50) not null,  
    CategorySchema UNIQUEIDENTIFIER null,  
    primary key (Id)  
)
```

2. 设置外键 Foreign Keys

在映射文件设置外键名称，注意有的需要两边都要设置才生效，代码片段如下：

```
<many-to-one name="CategorySchema"  
    class="DomainModel.Entities.CategorySchema,Do  
mainModel"  
    foreign-key="FK_Product_Category"/>
```

生成语句如下：

```
if exists (select 1 from sys.objects  
    where object_id = OBJECT_ID(N'[FK_Product_Cate  
gory]')  
    AND parent_object_id = OBJECT_ID('ProductSchem  
a'))  
alter table ProductSchema drop constraint FK_Product_Categ  
ory  
...  
alter table ProductSchema add constraint FK_Product_Categor  
y  
foreign key (CategorySchema) references CategorySchema
```

3. 设置 Unique 约束

我们要求 Name 字段唯一，添加 Unique 约束，代码片段如下：

```
<property name="Name" not-null="true" length="50" unique="true"/>
```

生成语句如下:

```
create table ProductSchema (  
  Id UNIQUEIDENTIFIER not null,  
  Name NVARCHAR(50) not null unique,  
  CategorySchema UNIQUEIDENTIFIER null,  
  primary key (Id)  
)
```

还有一种 unique-key 约束, 同时为两个属性设置 unique-key 约束。我们为 Customer 持久化类的 FirstName 和 LastName 属性添加 Unique 约束:

编写映射, 设置 FirstName 和 LastName 的 Unique 约束为 UK_Person_Name:

```
<property name="FirstName" not-null="true"  
  length="50" unique-key="UK_Customer_Name"/>  
<property name="LastName" not-null="true"  
  length="50" unique-key="UK_Customer_Name"/>
```

生成语句如下:

```
create table Customer(  
  CustomerId INT IDENTITY not null,  
  FirstName NVARCHAR(50) not null,  
  LastName NVARCHAR(50) not null,  
  primary key (CustomerId),  
  unique (FirstName, LastName)  
)
```

4. 设置索引 Index

```
<property name="Name" not-null="true" length="50"  
  unique="true" index="IDX_Product_Name" />
```

生成语句如下:

```
create table ProductSchema (Id UNIQUEIDENTIFIER...)  
create table CategorySchema (Id UNIQUEIDENTIFIER...)  
create index IDX_Product_Name on ProductSchema (Name)
```

5.设置 Check 约束

我们为 UnitsOnStock 值设置大于等于 0:

```
<property name="UnitsOnStock" not-null="true" >
  <column name="UnitsOnStock" check="UnitsOnStock >= 0"
/>
</property>
```

生成语句如下:

```
create table ProductSchema (
  Id UNIQUEIDENTIFIER not null,
  Name NVARCHAR(50) not null unique,
  UnitsOnStock INT null check( UnitsOnStock >= 0) ,
  CategorySchema UNIQUEIDENTIFIER null,
  primary key (Id)
)
```

好了, 还有很多设置大家自己探索啦! 知道了这些我们在写映射的时候就注意啦! 我之前映射文件属性映射为什么写的比较全呢? 就是这个原因, 便于生成数据库架构是自己约束的并不是默认的。

2.存储过程、视图

除了表, 我们还有存储过程和视图。怎么利用 SchemaExport 工具生成存储过程和视图呢? 在映射文件中提供了 database-object 元素用来创建和删除数据库对象。

```
<database-object>
  <create>创建存储过程或视图语句等数据库对象</create>
  <drop>删除存储过程或视图语句等数据库对象</drop>
</database-object>
```

1.存储过程

还记得我们在 [NHibernate 之旅\(17\): 探索 NHibernate 中使用存储过程\(下\)](#)中创建的三个存储过程了吗? 当时我们在数据库中手写创建的啊。现在完全可以不用那种古老的方法啦。我们在映射文件中编写 database-object: 在创建数据库架构时创建名为 entitySProcs 的存储过程, 在删除数据库架构时删除名为 entitySProcs 的存储过程。现在使用 SchemaExport 工具不仅仅生成了表, 还生成了存储过程。还剩两个存储过程就留给大家去完成吧。

```

<database-object>
  <create>
    CREATE PROCEDURE entitySProcs
    AS
    SELECT CustomerId,Version,Firstname,Lastname FROM Customer
  </create>
  <drop>
    DROP PROCEDURE entitySProcs
  </drop>
</database-object>

```

2.视图

还记得我们在 [NHibernate 之旅\(14\): 探索 NHibernate 中使用视图](#) 中创建了第一个视图了吗？那时我们也是在数据库中手动创建的，现在我们可以自动创建了！打开 CustomerView.hbm.xml 文件，添加 database-object：在创建数据库架构时创建名为 viewCustomer 的视图，在删除数据库架构时删除名为 viewCustomer 的视图。

```

<database-object>
  <create>
    CREATE VIEW [dbo].[viewCustomer]
    AS
    SELECT DISTINCT c.CustomerId, c.Firstname, c.Lastname,
    o.OrderId, o.OrderDate
    FROM      dbo.Customer AS c INNER JOIN
    dbo.[Order] AS o ON c.CustomerId = o.OrderId INNER JOIN
    N
    dbo.OrderProduct AS op ON o.OrderId = op.[Order] INNER
    JOIN
    dbo.Product AS p ON op.Product = p.ProductId
    GROUP BY c.CustomerId, c.Firstname, c.Lastname, o.OrderId, o.OrderDate
  </create>
  <drop>drop view dbo.viewCustomer</drop>
</database-object>

```

测试生成数据库架构，出现错误“数据库中已存在名为'viewCustomer'的对象”。这是什么原因呢？看看 NHibernate 生成语句：

```
create table viewCustomer (  
    CustomerId INT IDENTITY NOT NULL,  
    Firstname NVARCHAR(255) null,  
    Lastname NVARCHAR(255) null,  
    OrderId INT null,  
    OrderDate DATETIME null,  
    primary key (CustomerId)  
)
```

观察 NHibernate 生成 SQL 语句发现 NHibernate 利用 Class 映射自动生成了 viewCustomer 表，因为 NHibernate 见到 Class 映射就认为是表，它不知道这里映射的是视图，视图和表在映射文件中没有什么区别。我们修改一下这个映射文件，在 database-object 元素上面再添加一个 database-object 用于删除 NHibernate 生成的表。

```
<database-object>  
    <create>drop table dbo.viewCustomer</create>  
    <drop>drop table dbo.viewCustomer</drop>  
</database-object>
```

这个 database-object 的意思就是在创建数据库架构时删除 NHibernate 自动生成的表 viewCustomer，在删除数据库架构时删除表 viewCustomer。为什么删除两次呢？因为我们有可能只 Drop，那么 NHibernate 就执行 database-object 中的 Drop。

这样在 CustomerView.hbm.xml 文件中就有两个 database-object 了，总体的意思就是在创建数据库架构时删除 NHibernate 自动生成的表 viewCustomer 并创建名为 viewCustomer 的视图，在删除数据库架构时删除名为 viewCustomer 表和名为 viewCustomer 的视图。

结语

好了，终于把 SchemaExport 工具研究透彻了，应该没有说漏别的东西，就到这里了。下面看看 NHibernate 对象去，什么状态啦，缓存啦。

NHibernate 之旅(21): 探索对象状态

本节内容

- 引入
- 对象状态
- 对象状态转换
- 结语

引入

在程序运行过程中使用对象的方式对数据库进行操作，这必然会产生一系列的持久化类的实例对象。这些对象可能是刚刚创建并准备存储的，也可能是从数据库中查询的，为了区分这些对象，根据对象和当前会话的关联状态，我们可以把对象分为三种：

瞬时对象：对象刚刚建立。该对象在数据库中没有记录，也不在 `ISession` 缓存中。如果该对象是自动生成主键，则该对象的对象标识符为空。

持久化对象：对象已经通过 `NHibernate` 进行了持久化，数据库中已经存在对应的记录。如果该对象是自动生成主键，则该对象的对象标识符已被赋值。

托管对象：该对象是经过 `NHibernate` 保存过或者从数据库中取出的，但是与之关联的 `ISession` 已经关闭。虽然它有对象标识符且数据库中存在对应记录，但是已经不再被 `NHibernate` 管理。

对象状态

`NHibernate` 提供了对象状态管理的功能，支持三种对象状态：瞬时态(`Transient`)、持久态(`Persistent`)、托管态(`Detached`)。

1. 瞬时态(`Transient`)

对象刚刚创建，还没有来及和 `ISession` 关联的状态。这时瞬时对象不会被持久化到数据库中，也不会被赋上标识符。如果不使用则被 `GC` 销毁。`ISession` 接口可以将其转换为持久状态。

这像这样，刚刚创建了一个 `Customer` 对象，是一个瞬时态对象：

```
var customer = new Customer() { Firstname = "YJing", Lastname = "Lee" };
```

2.持久态(Persistent)

刚被保存的或刚从数据库中加载的。对象仅在相关联的 `ISession` 生命周期内有效，在数据库中有相应记录并有标识符。对象实例由 `NHibernate` 框架管理，如果有任何改动，在当然操作提交时，与数据库同步，即将对象保存更新到数据库中。

3.托管态(Detached)

持久对象关联的 `ISession` 关闭后，这个对象在 `ISession` 中脱离了关系，就是托管态了，托管对象仍然有持久对象的所有属性，对托管对象的引用仍然有效的，我们可以继续修改它。如果把这个对象重新关联到 `ISession` 上，则再次转变为持久态，在托管时期的修改会被持久化到数据库中。

对象状态转换

在同步数据库的情况下执行下面的语句可以转换对象的状态。

测试验证对象

`ISession.Contains(object)`: 检查 `ISession` 中是否包含指定实例

重新设置 `ISession`

```
private void ResetSession()
{
    if (_session.IsOpen)
        _session.Close();
    _session = _sessionManager.GetSession();
    _transaction.Session = _session;
}
```

1.瞬时态转换持久态

方法一: `ISession.Save()`: 保存指定实例。

```
[Test]
public void TransientConvertPersistentTest()
{
    //瞬时态对象
```



```

var customer = new Customer() { Firstname = "YJidng", Lastname = "Lee" };
Assert.IsFalse(_session.Contains(customer));
//仍然是瞬时态, CustomerId 属性值为空

//关联 ISession 保存到数据库中
_session.Save(customer);
//变为持久态, 由于表中 CustomerId 字段自动增长的, 保存数据库, CustomerId 字段自动加一
//经过 NHibernate 类型转换后返回 CustomerId 属性值, 保证数据库与实例对象同步
Assert.IsTrue(_session.Contains(customer));
}

```

方法二: `ISession.SaveOrUpdate()`: 分配新标识保存瞬时态对象。

2.持久态转换托管态

方法一: `ISession.Evict(object)`: 从当前 `ISession` 中删除指定实例

```

[Test]
public void PersistentConvertDetachedEvictTest()
{
    Customer customer = _transaction.GetCustomerById(1);
    Assert.IsTrue(_session.Contains(customer));
    _session.Evict(customer);
    Assert.IsFalse(_session.Contains(customer));
}

```

方法二: `ISession.Close()`: 关闭当前 `ISession`

```

[Test]
public void PersistentConvertDetachedCloseTest()
{
    Customer customer = _transaction.GetCustomerById(1);
    Assert.IsTrue(_session.Contains(customer));
    ResetSession();
    Assert.IsFalse(_session.Contains(customer));
}

```

3.托管态转换持久态

方法一：ISession.Update(): 更新指定实例。

```
[Test]
public void DetachedConvertPersistentUpdateTest()
{
    Customer customer = _transaction.GetCustomerById(1);
    //持久态对象
    Assert.IsTrue(_session.Contains(customer));
    //重新设置 ISession
    ResetSession();
    Assert.IsFalse(_session.Contains(customer));
    //托管态对象
    //在托管态下可继续被修改
    customer.Firstname += "CnBlogs";
    _transaction.UpdateCustomerTransaction(customer);
    //转变为持久态对象
    Assert.IsTrue(_session.Contains(customer));
}
```

看看这个例子：在托管时期的修改会被持久化到数据库中；

注意：NHibernate 如何知道重新关联的对象是不是“脏的(修改过的)”？如果是新的 ISession，ISession 就不能与对象初值来比较这个对象是不是“脏的”，我们在映射文件中定义<id>元素和<version>元素的 unsaved-value 属性，NHibernate 就可以自己判断了。

```
[Test]
public void DetachedConvertPersistentUpdateAllTest()
{
    Customer customer = _transaction.GetCustomerById(1);
    //持久态对象
    customer.Firstname += "YJingLee";
    Assert.IsTrue(_session.Contains(customer));
    //重新设置 ISession
    ResetSession();
    Assert.IsFalse(_session.Contains(customer));
    //托管态对象
    //在托管态下可继续被修改
    customer.Firstname += "CnBlogs";
    //这时一起更新
    _transaction.UpdateCustomerTransaction(customer);
}
```

```
//转变为持久态对象
Assert.IsTrue(_session.Contains(customer));
}
```

这个加上一个锁：如果在托管时期没有修改，就不执行更新语句，只转换为持久态，下面的例子如果在托管时期修改对象，执行更新语句。

```
[Test]
public void DetachedConvertPersistentUpdateLockTest()
{
    Customer customer = _transaction.GetCustomerById(1);
    Assert.IsTrue(_session.Contains(customer));
    ResetSession();
    Assert.IsFalse(_session.Contains(customer));
    //锁
    _session.Lock(customer, NHibernate.LockMode.None);
    //如果在托管时期没有修改，就不执行更新语句，只转换为持久态
    //customer.Firstname += "CnBlogs";
    _transaction.UpdateCustomerTransaction(customer);
    Assert.IsTrue(_session.Contains(customer));
}
```

方法二：`ISession.Merge()`：合并指定实例。不必考虑 `ISession` 状态，`ISession` 中存在相同标识的持久化对象时，`NHibernate` 便会根据用户给出的对象状态覆盖原有的持久化实例状态。

方法三：`ISession.SaveOrUpdate()`：分配新标识保存瞬时态对象；更新/重新关联托管态对象。

以上两个大家自己测试了！

结语

这篇初步知道了对象的状态。虽然对象的状态的细节由 `NHibernate` 自己维护，但是对象状态在 `NHibernate` 应用中还是比较重要的。同时对象状态也涉及了 `NHibernate` 缓存、离线查询等内容。