

引言

在软件开发过程中，并发控制是确保及时纠正由并发操作导致的错误的一种机制。从 ADO.NET 到 LINQ to SQL 再到如今的 ADO.NET Entity Framework，.NET 都为并发控制提供好良好的支持方案。

并发处理方式一般分为乐观必并发与悲观必并发两种，本文将为大家介绍 Entity Framework 、 LINQ to SQL 中的并发处理方式。在本文最后，将提供一个了可参考的方案，结合事务 Transaction 处理复杂性对象的并发。

目录

- 一、并发处理的定义
- 二、模型属性的并发处理选项
- 三、Entity Framewrok 悲观并发
- 四、Entity Framework 乐观并发
- 五、回顾 LINQ to SQL 并发处理的方式
- 六、结合事务处理并发冲突

一、并发处理的定义

在软件开发过程中，当多个用户同时修改一条数据记录时，系统需要预先制定对并发的处理模式。并发处理模式主要分为两种：第一种模式称为悲观式并发，即当一个用户已经在修改某条记录时，系统将拒绝其他用户同时修改此记录。

第二种模式称为乐观式并发，即系统允许多个用户同时修改同一条记录，系统会预先定义由数据并发所引起的并发异常处理模式，去处理修改后可能发生的冲突。常用的乐观性并发处理方法有以下几种：

- 1. 保留最后修改的值。
- 2. 保留最初修改的值。
- 3. 合并多次修改的值。

相对于LINQ TO SQL 中的并发处理方式，Entity Framework 中的并发处理方式实现了不少的简化，下面为大家一一介绍。

[回到目录](#)

二、模型属性的并发处理选项

在System.Data.Metadata.Edm 命名空间中，存在ConcurrencyMode 枚举，用于指定概念模型中的属性的并发选项。ConcurrencyMode 有两个成员：

成员名称	说明
None	在写入时从不验证此属性。 这是默认的并发模式。
Fixed	在写入时始终验证此属性。

当模型属性为默认值 None 时，系统不会对此模型属性进行检测，当同一个时间对此属性进行修改时，系统会以数据合并方式处理输入的属性值。

当模型属性为Fixed 时，系统会对此模型属性进行检测，当同一个时间对属性进行修改时，系统就会激发 OptimisticConcurrencyException 异常。

开发人员可以为对象的每个属性定义不同的 ConcurrencyMode 选项，选项可以在*.csdl 中看到：

```
1 <Schema>
2     .....
3     .....
4 <EntityType Name="Person">
5     <Key>
6         <PropertyRef Name="Id" />
7     </Key>
8     <Property Type="Int32" Name="Id" Nullable="false" annotation:StoreGeneratedPattern="Identity" />
9     <Property Type="String" Name="FirstName" MaxLength="50" FixedLength="false" Unicode="true"
10         ConcurrencyMode="Fixed" />
11     <Property Type="String" Name="SecondName" MaxLength="50" FixedLength="false" Unicode="true" />
12     <Property Type="Int32" Name="Age" />
13     <Property Type="String" Name="Address" MaxLength="50" FixedLength="false" Unicode="true" />
14     <Property Type="String" Name="Telephone" MaxLength="50" FixedLength="false" Unicode="true" />
15     <Property Type="String" Name="EMail" MaxLength="50" FixedLength="false" Unicode="true" />
16 </EntityType>
17 </Schema>
```

02年毕业于中山大学。从事.NET、JAVA的系统架构。对领域驱动业务架构SOA、分布式相互调用等方面有着的文章皆属原创，转



技术交流
欢迎加入以下小组共
博客园小组：
.NET高级编程
QQ群：
.NET 高级编程 230
JAVA 高级编程 174

昵称：风尘浪子
园龄：7年1个月
荣誉：推荐博客
粉丝：1505
关注：11
+加关注

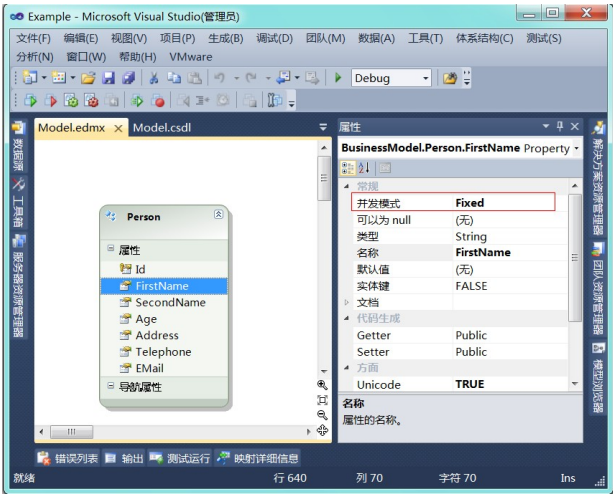
搜索

- 随笔分类(49)
- .NET基础篇(6)
 - Android开发笔记(3)
 - ASP.NET基础篇(3)
 - C#综合揭秘(7)
 - Javascript摘要(1)
 - Java与.NET的相互调
 - Java远程通讯技术(2)
 - SOA面向服务架构(3)
 - WCF揭秘(5)
 - WF与Web服务的相互
 - 程序人生(3)
 - 服务类工具的应用与
 - 利用远程对象实现分
 - 软件工具(3)
 - 项目管理(1)

Android开发笔记
Cordova 5.3 搭建 A
深入剖析 Android S
通过CordovaPlugin

C#综合揭秘
Entity Framework
深入分析委托与事件
通过修改注册表建立
细说多线程（上）
细说多线程（下）
细说进程、应用程序
细说事务

JAVA与.NET的相
TCP/IP相互调用基本
利用JNBridge实现远
通过Web服务实现相



[回到目录](#)

三、Entity Framework 悲观并发

在一般的开发过程中，最常用的是悲观并发处理。.NET 提供了Lock、Monitor、Interlocked 等多个锁定数据的方式，它可以保证同一个表里的对象不会同时被多个客户进行修改，避免了系统数据出现逻辑性的错误。由于本篇文章主要介绍并发处理方式，关于锁的介绍，请参考<http://www.cnblogs.com/leslies2/archive/2012/02/08/2320914.html#t8>

```
1 private static object o=new object();
2
3 public int Update(Person person)
4 {
5     int n = -1;
6     try
7     {
8         lock (o)
9         {
10             using (BusinessEntities context = new BusinessEntities())
11             {
12                 var obj = context.Person.Where(x => x.Id == person.Id).First();
13                 if (obj != null)
14                     context.ApplyCurrentValues("Person", person);
15                 n = context.SaveChanges();
16             }
17         }
18     }
19     catch (Exception ex)
20     { ..... }
21     return n;
22 }
```

使用悲观并发虽然能有效避免数据发生逻辑性的错误，但使用 lock 等方式锁定 Update 方法的操作，在用户同时更新同一数据表的数据，操作就会被延时或禁止。在千万级 PV 的大型网络系统当中使用悲观并发，有可能降低了系统的效率，此时可以考虑使用乐观并发处理。

[回到目录](#)

四、Entity Framework 乐观并发

为了解决悲观并发所带来的问题，ADO.NET Entity Framework 提供了更为高效的乐观并发处理方式。相对于LINT to SQL , ADO.NET Entity Framework 简化了乐观并发的处理方式，它可以灵活使用合并数据、保留初次输入数据、保留最新输入数据等方式处理并发冲突。

4.1 以合并方式处理并发数据

当模型属性的 ConcurrencyMode 为默认值 None ，一旦同一个对象属性同时被修改，系统将以合并数据的方式处理并发冲突，这也是 Entity Framework 处理并发冲突的默认方式。合并处理方式如下：当同一时间针对同一个对象属性作出修改，系统将保存最新输入的属性值。当同一时间对同一对象的不同属性作出修改，系统将保存已被修改的属性值。下面用两个例子作出说明：

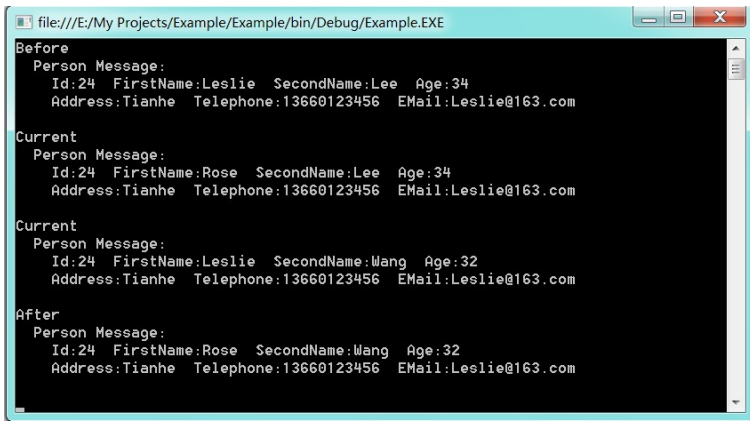
4.1.1 同时更新数据

在系统输入下面代码，获取数据库中的Person: Id 为24，FirstName为Leslie，SecondName为Lee。然后使用异步方法分两次调用 Update方法，同时更新Person对象的相关属性，第一次更新把对象的 FirstName 属性改Rose，第二次更新把对象的 SecondName改为Wang，Age改为32。在使用SaveChanges保存更新时，第一个方法已经把ObjectContext1中的FirstName修改为Rose，但在第二个方法中的ObjectContext2中的FirstName依然是Leslie，此时由于三个属性的ConcurrencyMode都为默认值None，系统会忽略当中的冲突，而接受所有的更新修改。

```
1 public class PersonDAL
2 {
3     public Person GetPerson(int id)
```

```
4      {
5          using (BusinessEntities context = new BusinessEntities())
6          {
7              IQueryable<Person> list=context.Person.Where(x => x.Id == id);
8              return list.First();
9          }
10     }
11
12     public void Update(Person person)
13     {
14         using (BusinessEntities context = new BusinessEntities())
15         {
16             //显示输入新数据的信息
17             Display("Current", person);
18             var obj = context.Person.Where(x => x.Id == person.Id).First();
19             if (obj != null)
20                 context.ApplyCurrentValues("Person", person);
21
22             //虚拟操作, 保证数据能同时加入到上下文当中
23             Thread.Sleep(100);
24             context.SaveChanges();
25         }
26     }
27
28     delegate void MyDelegate(Person person);
29
30     public static void Main(string[] args)
31     {
32         //在更新数据前显示对象信息
33         PersonDAL personDAL = new PersonDAL();
34         var beforeObj = personDAL.GetPerson(24);
35         personDAL.Display("Before", beforeObj);
36
37         //更新Person的SecondName, Age两个属性
38         Person person1 = new Person();
39         person1.Id = 24;
40         person1.FirstName = "Leslie";
41         person1.SecondName = "Wang";
42         person1.Age = 32;
43         person1.Address = "Tianhe";
44         person1.Telephone = "13660123456";
45         person1.Email = "Leslie@163.com";
46
47         //更新Person的FirstName属性
48         Person person2 = new Person();
49         person2.Id = 24;
50         person2.FirstName = "Rose";
51         person2.SecondName = "Lee";
52         person2.Age = 34;
53         person2.Address = "Tianhe";
54         person2.Telephone = "13660123456";
55         person2.Email = "Leslie@163.com";
56
57         //使用异步方式同时更新数据
58         MyDelegate myDelegate = new MyDelegate(personDAL.Update);
59         myDelegate.BeginInvoke(person1, null, null);
60         myDelegate.BeginInvoke(person2, null, null);
61
62         Thread.Sleep(300);
63         //在更新数据后显示对象信息
64         var afterObj = personDAL.GetPerson(24);
65         personDAL.Display("After", afterObj);
66         Console.ReadKey();
67     }
68
69     public void Display(string message, Person person)
70     {
71         String data = string.Format("{0}\n Person Message:\n Id:{1} FirstName:{2} "+
72             "SecondName:{3} Age:{4}\n Address:{5} Telephone:{6} Email:{7}\n",
73             message, person.Id, person.FirstName, person.SecondName, person.Age,
74             person.Address, person.Telephone, person.Email);
75         Console.WriteLine(data);
76     }
77 }
```

根据操作结果可以看到, 在Entity Framework的默认环境情况下, 系统会使用合并方式处理并发, 把输入数据的所有修改值都保存到当前上下文当中, 并同时修改数据库当中的值。



4.1.2 删除与更新操作同时运行

Entity Framework 能以完善的机制灵活处理同时更新同一对象的操作，但一旦删除操作与更新操作同时运行时，就可能存在逻辑性的异常。例如：两个客户端同时加载了同一个对象，第一个客户端更新了数据后，把数据再次提交。但在提交前，第二个客户端已经把数据库中的已有数据删除。此时，上下文中的对象处于不同的状态底下，将会引发 `OptimisticConcurrencyException` 异常。

遇到此异常时，可以用 `try (OptimisticConcurrencyException) {...} catch {...}` 方式捕获异常，然后使用 `ObjectStateManager.ChangeObjectState` 方法更改对象的 `EntityState` 属性。把 `EntityState` 更改为 `Added`，被删除的数据便会被再次加载。若把 `EntityState` 更改为 `Detached` 时，数据便会被顺利删除。下面把对象的 `EntityState` 属性更改为 `Added` 作为例子。

```
1 public class PersonDAL
2 {
3     delegate int MyDelegate(Person person);
4
5     public static void Main(string[] args)
6     {
7         //在更新数据前显示对象信息
8         PersonDAL personDAL = new PersonDAL();
9         var beforeObj = personDAL.GetPerson(51);
10        personDAL.DisplayProperty("Begin", beforeObj);
11
12        //更新Person的属性
13        Person person1 = new Person();
14        person1.Id = 51;
15        person1.FirstName = "Leslie";
16        person1.SecondName = "Wang";
17        person1.Age = 32;
18        person1.Address = "Tianhe";
19        person1.Telephone = "13660123456";
20        person1.EMail = "Leslie@163.com";
21
22        //使用异步方式更新数据
23        MyDelegate myDelegate = new MyDelegate(personDAL.Update);
24        IAsyncResult reslut=myDelegate.BeginInvoke(person1, null, null);
25
26        //同步删除原有数据
27        personDAL.Delete(51);
28        //显示删除后重新被加载的数据
29        var afterObj = personDAL.GetPerson(myDelegate.EndInvoke(reslut));
30        personDAL.DisplayProperty("End", afterObj);
31    }
32
33    public Person GetPerson(int id)
34    {
35        using (BusinessEntities context = new BusinessEntities())
36        {
37            IQueryable<Person> list=context.Person.Where(x => x.Id == id);
38            return list.First();
39        }
40    }
41
42    //更新对象
43    public int Update(Person person)
44    {
45        int returnValue=-1;
46        using (BusinessEntities context = new BusinessEntities())
47        {
48            var obj = context.Person.Where(x => x.Id == person.Id).First();
49            //显示对象所处状态
50            DisplayState("Before Update", obj);
51            try
52            {
53                if (obj != null)
54                    context.ApplyCurrentValues("Person", person);
55                //虚拟操作，保证数据已经在数据库中被异步删除
56                Thread.Sleep(100);
57                context.SaveChanges();
58            }
59            catch (OptimisticConcurrencyException)
60            {
61                //这里处理异常
62            }
63        }
64        return returnValue;
65    }
66
67    private void DisplayState(string state, Person obj)
68    {
69        Console.WriteLine("EntityState: {0}, Person: {1}", obj.EntityState, obj);
70    }
71}
```

```
58         returnValue = obj.Id;
59     }
60     catch (System.Data.OptimisticConcurrencyException ex)
61     {
62         //把对象的状态更改为 Added
63         context.ObjectStateManager.ChangeObjectState(obj, System.Data.EntityState.Added);
64         context.SaveChanges();
65         returnValue=obj.Id;
66     }
67 }
68 return returnValue;
69 }
70
71 //删除对象
72 public void Delete(int id)
73 {
74     using (BusinessEntities context = new BusinessEntities())
75     {
76         var person1 = context.Person.Where(x => x.Id == id).First();
77         if (person1 != null)
78             context.Person.DeleteObject(person1);
79         context.SaveChanges();
80         //显示对象现在所处的状态
81         DisplayState("After Delete:", person1);
82     }
83 }
84
85 //显示对象现在所处的状态
86 public void DisplayState(string message, Person person)
87 {
88     String data = string.Format("{0}\n Person State:{1}\n",
89         message, person.EntityState);
90     Console.WriteLine(data);
91 }
92 //显示对象相关属性
93 public void DisplayProperty(string message, Person person)
94 {
95     String data = string.Format("{0}\n Person Message:\n Id:{1} FirstName:{2} " +
96         "SecondName:{3} Age:{4}\n Address:{5} Telephone:{6} EMail:{7}\n",
97         message, person.Id, person.FirstName, person.SecondName, person.Age,
98         person.Address, person.Telephone, person.EMail);
99     Console.WriteLine(data);
100 }
101 }
```

观察运行测试结果，当运行 `Delete` 方法，对象已经在数据库中被删除，对象的 `EntityState` 处于 `Detached` 状态。此时使用 `SaveChanges` 保存更新数据时，引发了 `OptimisticConcurrencyException` 异常。在捕获异常，把对象状态更改为 `Added`，再使用 `SaveChanges` 保存数据，数据就能顺利地保存到数据库中。但值得留意，因为对象是在删除后重新加载的，所以对象的 `Id` 也会被同步更新。

```
file:///E:/My Projects/Example/Example/bin/Debug/Example.EXE
Begin
  Person Message:
  Id:51 FirstName:Mike SecondName:Chen Age:28
  Address:BeijingRoad Telephone:13425636589 EMail:Mike@163.com

Before Update
  Person State:Unchanged

After Delete:
  Person State:Detached

End
  Person Message:
  Id:52 FirstName:Leslie SecondName:Wang Age:32
  Address:Tianhe Telephone:13660123456 EMail:Leslie@163.com
```

以合并数据的方式处理并发冲突固然方便快捷，但在业务逻辑较为复杂的系统下并不适合使用此处理方式。比如在常见的 `Order`、`OrderItem` 的表格中，`OrderItem` 的单价，数量会直接影响 `Order` 的总体价格，这样使用合并数据的方式处理并发，有可能引起逻辑性的错误。此时，应该考虑以其他方式处理并发冲突。

4.2 当发生数据并发时，保留最新输入的数据

要验证输入对象的属性，必须先把该属性的 `ConcurrencyMode` 设置为 `Fixed`，这样系统就会实时检测对象属性的输入值。当该属性被同时更新，系统便会激发 `OptimisticConcurrencyException` 异常。捕获该异常后，可以使用 `ObjectContext.Refresh(RefreshMode, object)` 刷新上下文中该对象的状态，当 `RefreshMode` 为 `ClientWins` 时，系统将会保持上下文中的现有数据，即保留最新输入的对象值。此时再使用 `ObjectContext.SaveChanges`，系统就会把最新输入的对象值加入数据库当中。

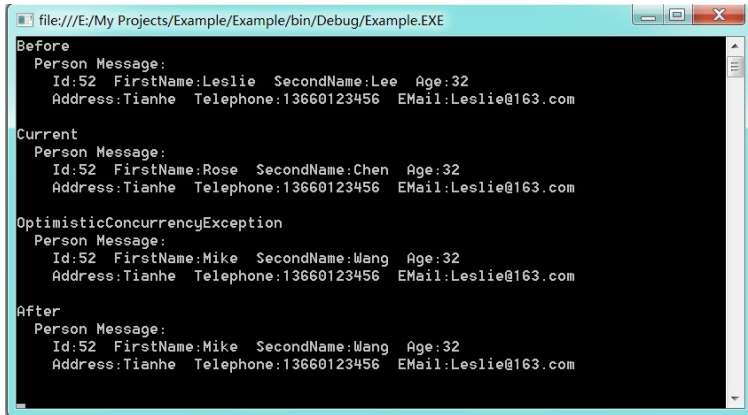
在下面的例子当，系统启动前先把 `Person` 的 `FirstName`、`SecondName` 两个属性的 `ConcurrencyMode` 属性设置为 `Fixed`，使系统能监视这两个属性的更改。所输入的数据只在 `FirstName`、`SecondName` 两个值中作出修改。在数据提交前先以 `DisplayProperty` 方法显示数据库最初的数据属性，在数据初次更新后再调用 `DisplayProperty` 显示更新后的数据属性。在第二次更新数据时，由调用 `ObjectContext.SaveChanges` 时，数据库中的数据已经被修改，与当前上下文 `ObjectContext` 的数据存在冲突，系统将激发 `OptimisticConcurrencyException` 异常，此时把引发异常的对象属性再次显示出来。对异常进行处理后，显示数据库中最终的对象值。

```
1 public class PersonDAL
```

```
2      {
3          delegate void MyDelegate(Person person);
4
5          public static void Main(string[] args)
6          {
7              //在更新数据前显示对象信息
8              PersonDAL personDAL = new PersonDAL();
9              var beforeObj = personDAL.GetPerson(52);
10             personDAL.DisplayProperty("Before", beforeObj);
11
12             //更新Person的FirstName、SecondName属性
13             Person person1 = new Person();
14             person1.Id = 52;
15             person1.FirstName = "Mike";
16             person1.SecondName = "Wang";
17             person1.Age = 32;
18             person1.Address = "Tianhe";
19             person1.Telephone = "13660123456";
20             person1.Email = "Leslie@163.com";
21
22             //更新Person的FirstName、SecondName属性
23             Person person2 = new Person();
24             person2.Id = 52;
25             person2.FirstName = "Rose";
26             person2.SecondName = "Chen";
27             person2.Age = 32;
28             person2.Address = "Tianhe";
29             person2.Telephone = "13660123456";
30             person2.Email = "Leslie@163.com";
31
32             //使用异步方式更新数据
33             MyDelegate myDelegate = new MyDelegate(personDAL.Update);
34             myDelegate.BeginInvoke(person1, null, null);
35             myDelegate.BeginInvoke(person2, null, null);
36             //显示完成更新后数据源中的对应属性
37             Thread.Sleep(1000);
38             var afterObj = personDAL.GetPerson(52);
39             personDAL.DisplayProperty("After", afterObj);
40         }
41
42         public Person GetPerson(int id)
43         {
44             using (BusinessEntities context = new BusinessEntities())
45             {
46                 IQueryable<Person> list=context.Person.Where(x => x.Id == id);
47                 return list.First();
48             }
49         }
50
51         //更新对象
52         public void Update(Person person)
53         {
54             using (BusinessEntities context = new BusinessEntities())
55             {
56                 var obj = context.Person.Where(x => x.Id == person.Id).First();
57                 try
58                 {
59                     if (obj!=null)
60                         context.ApplyCurrentValues("Person", person);
61                     //虚拟操作，保证数据被同步加载
62                     Thread.Sleep(100);
63                     context.SaveChanges();
64                     //显示第一次更新后的数据属性
65                     this.DisplayProperty("Current", person);
66                 }
67                 catch (System.Data.OptimisticConcurrencyException ex)
68                 {
69                     //显示发生OptimisticConcurrencyException异常所输入的数据属性
70                     this.DisplayProperty("OptimisticConcurrencyException", person);
71
72                     if (person.EntityKey == null)
73                         person.EntityKey = new System.Data.EntityKey("BusinessEntities.Person",
74                             "Id", person.Id);
75                     //保持上下文当中对象的现有属性
76                     context.Refresh(RefreshMode.ClientWins, person);
77                     context.SaveChanges();
78                 }
79             }
80         }
81
82         //显示对象相关属性
83         public void DisplayProperty(string message, Person person)
84         {
85             String data = string.Format("{0}\n Person Message:\n Id:{1} FirstName:{2} " +
86                 "SecondName:{3} Age:{4}\n Address:{5} Telephone:{6} Email:{7}\n",
87                 message, person.Id, person.FirstName, person.SecondName, person.Age,
88                 person.Address, person.Telephone, person.Email);
```

```
89         Console.WriteLine(data);
90     }
91 }
```

观察测试结果，可见当RefreshMode状态为ClientWins时，系统将会保存上下文当中的对象属性，使用此方法可以在发生并发异常时保持最新输入的对象属性。



4.3 当发生数据并发时，保留最初输入的数据

把对象属性的 ConcurrencyMode 设置为 Fixed 后，同时更新该属性，将会激发 OptimisticConcurrencyException 异常。此时使用ObjectContext.Refresh (RefreshMode,object) 刷新上下文中该对象的状态，当 RefreshMode 为 StoreWins 时，系统就会把数据源中的数据代替上下文中的数据。

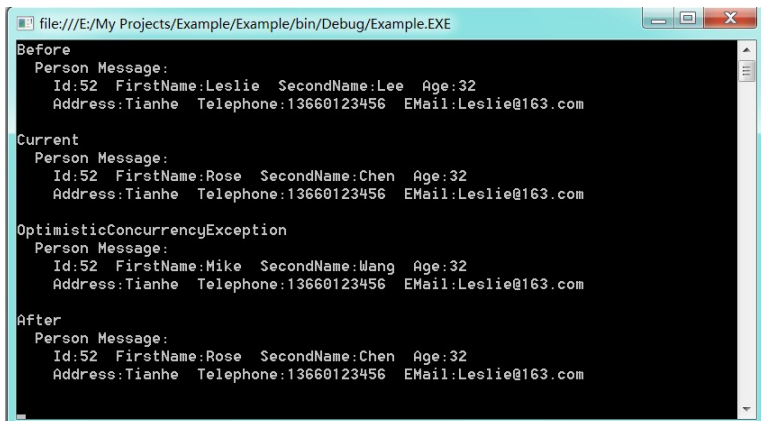
因为初次调用 SaveChanges，数据可以成功保存到数据库。但是在 ObjectContext 并未释放时，再次使用 SaveChanges 异步更新数据，就会引发 OptimisticConcurrencyException 并发异常。当 RefreshMode 为 StoreWins 时，系统就会保留初次输入的数据属性。此例子与上面的例子十分相似，只是把 RefreshMode 改为 StoreWins 而已。在业务逻辑较为复杂的系统当中，建议使用此方式处理并发异常。在保留最初输入的数据修改属性后，把属性返还给客户，让客户进行对比后再决定下一步的处理方式。

```
1 public class PersonDAL
2 {
3     delegate void MyDelegate(Person person);
4
5     public static void Main(string[] args)
6     {
7         //在更新数据前显示对象信息
8         PersonDAL personDAL = new PersonDAL();
9         var beforeObj = personDAL.GetPerson(52);
10        personDAL.DisplayProperty("Before", beforeObj);
11
12        //更新Person的FirstName、SecondName属性
13        Person person1 = new Person();
14        person1.Id = 52;
15        person1.FirstName = "Mike";
16        person1.SecondName = "Wang";
17        person1.Age = 32;
18        person1.Address = "Tianhe";
19        person1.Telephone = "13660123456";
20        person1.EMail = "Leslie@163.com";
21
22        //更新Person的FirstName、SecondName属性
23        Person person2 = new Person();
24        person2.Id = 52;
25        person2.FirstName = "Rose";
26        person2.SecondName = "Chen";
27        person2.Age = 32;
28        person2.Address = "Tianhe";
29        person2.Telephone = "13660123456";
30        person2.EMail = "Leslie@163.com";
31
32        //使用异步方式更新数据
33        MyDelegate myDelegate = new MyDelegate(personDAL.Update);
34        myDelegate.BeginInvoke(person1, null, null);
35        myDelegate.BeginInvoke(person2, null, null);
36        //显示完成更新后数据源中的对应属性
37        Thread.Sleep(1000);
38        var afterObj = personDAL.GetPerson(52);
39        personDAL.DisplayProperty("After", afterObj);
40    }
41
42    public Person GetPerson(int id)
43    {
44        using (BusinessEntities context = new BusinessEntities())
45        {
46            IQueryable<Person> list=context.Person.Where(x => x.Id == id);
47            return list.First();
48        }
49    }
50 }
```



```
48     }
49 }
50
51 //更新对象
52 public void Update(Person person)
53 {
54     using (BusinessEntities context = new BusinessEntities())
55     {
56         var obj = context.Person.Where(x => x.Id == person.Id).First();
57         try
58         {
59             if (obj!=null)
60                 context.ApplyCurrentValues("Person", person);
61             //虚拟操作, 保证数据被同步加载
62             Thread.Sleep(100);
63             context.SaveChanges();
64             //显示第一次更新后的数据属性
65             this.DisplayProperty("Current", person);
66         }
67         catch (System.Data.OptimisticConcurrencyException ex)
68         {
69             //显示发生OptimisticConcurrencyException异常所输入的数据属性
70             this.DisplayProperty("OptimisticConcurrencyException", person);
71
72             if (person.EntityKey == null)
73                 person.EntityKey = new System.Data.EntityKey("BusinessEntities.Person",
74                     "Id", person.Id);
75             //保持数据源中对象的现有属性
76             context.Refresh(RefreshMode.StoreWins, person);
77             context.SaveChanges();
78         }
79     }
80 }
81
82 //显示对象相关属性
83 public void DisplayProperty(string message, Person person)
84 {
85     String data = string.Format("{0}\n Person Message:\n      Id:{1}  FirstName:{2}  " +
86         "SecondName:{3}  Age:{4}\n      Address:{5}  Telephone:{6}  EMail:{7}\n",
87         message, person.Id, person.FirstName, person.SecondName, person.Age,
88         person.Address, person.Telephone, person.Email);
89     Console.WriteLine(data);
90 }
91 }
```

观察测试结果, 可见当 **RefreshMode** 状态为 **StoreWins** 时, 系统将会以数据源中的数据代替上下文当中的对象属性。在业务逻辑较为复杂的系统中, 建议使用此方式处理并发异常。



```
file:///E:/My Projects/Example/Example/bin/Debug/Example.EXE
Before
Person Message:
Id:52  FirstName:Leslie  SecondName:Lee  Age:32
Address:Tianhe  Telephone:13660123456  EMail:Leslie@163.com

Current
Person Message:
Id:52  FirstName:Rose  SecondName:Chen  Age:32
Address:Tianhe  Telephone:13660123456  EMail:Leslie@163.com

OptimisticConcurrencyException
Person Message:
Id:52  FirstName:Mike  SecondName:Wang  Age:32
Address:Tianhe  Telephone:13660123456  EMail:Leslie@163.com

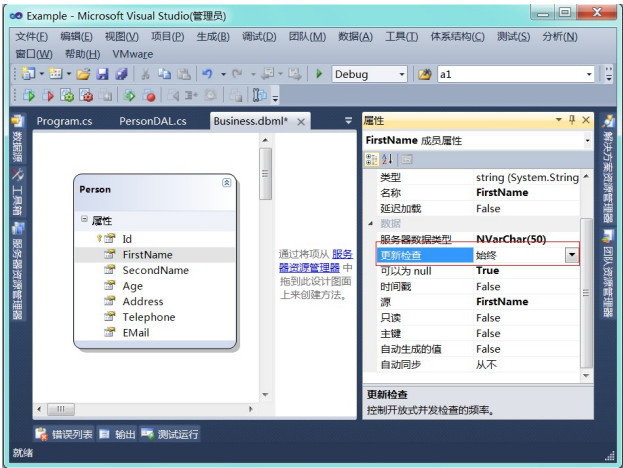
After
Person Message:
Id:52  FirstName:Rose  SecondName:Chen  Age:32
Address:Tianhe  Telephone:13660123456  EMail:Leslie@163.com
```

[回到目录](#)

五、回顾 LINQ to SQL 并发处理的方式

Entity Framework 当中简化了并发处理的方式, 然而温故而知新, LINQ to SQL 中并发处理所使用的方式也值得回顾一下。下面将与大家一起回顾一下 LINQ to SQL 当中并发处理的方式。

与 Entity Framework 相似, LINQ to SQL 中表格的每个列都为可以设定不同处理方式, 属性 **UpdateCheck** (更新检查) 的默认值为 **Always**, 即系统会在默认情况下检查属性的并发状态。若把属性改为 **WhenChanged**, 即当该属性发生改变时, 系统才会对其进行检测。若把属性改为 **Nerver**, 这时系统将不会对此属性进行检查, 总是接受最新一次的输入值。



处理 LINQ to SQL 并发，最为重要的是以下两个方法：

```
DataContext.SubmitChanges(ConflictMode)
DataContext.ChangeConflicts.ResolveAll(RefreshMode);
```

SubmitChanges 将对检索到的对象所做的更改发送到基础数据库，并通过 ConflictMode 指定并发冲突时要采取的操作。当选择 ConflictMode.FailOnFirstConflict 时，若检测到第一个并发冲突错误时，系统会立即停止对更新数据库的尝试。当选择 Conflict.ContinueOnConflict 时，系统会尝试运行对数据库的所有更新。

成员名称	说明
FailOnFirstConflict	指定当检测到第一个并发冲突错误时，应立即停止对更新数据库的尝试。
ContinueOnConflict	指定应尝试对数据库的所有更新，并且应在该过程结束时累积和返回并发冲突。

ConflictMode成员图

ResolveAll 能使用指定策略解决集合中的所有冲突，当选择 RefreshMode.KeepChanges 时，系统会强制 Refresh 方法保持当前上下文的数据值。当选择RefreshMode.OverwriteCurrentValues，系统会用数据库的值覆盖当前上下文中的数据值。当选择 RefreshMode.KeepCurrentValues，系统会把当前上下文的更新值与数据库中的值进行合并处理。

成员名称	说明
OverwriteCurrentValues	强制 Refresh 方法使用数据库中的值重写所有当前值。
KeepChanges	强制 Refresh 方法保留已更改的当前值，但将其他值更新为数据库值。
KeepCurrentValues	强制 Refresh 方法使用从数据库检索的值替换原始值。不会修改当前值。

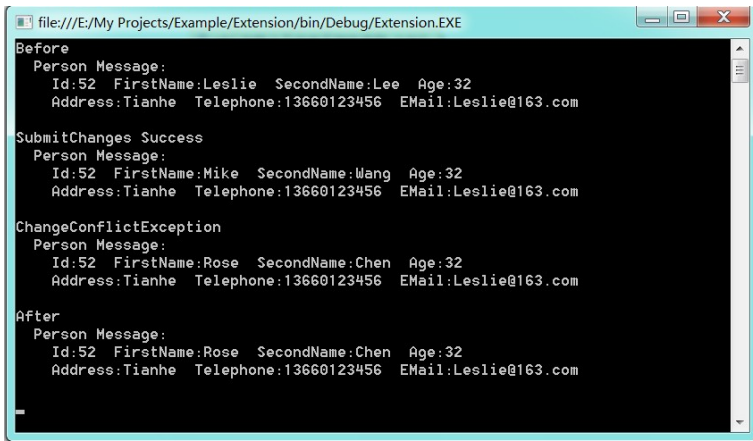
RefreshMode 成员图

当 Person 表格的多个列的 UpdateCheck 属性都为默认值 Always 时，多个客户同时更新此数据表，最后使用 DataContext.SubmitChanges(ConflictMode.ContinuOnConflict) 同时提交数据时，系统就会释放出 ChangeConflictException 异常。系统可以以捕获此并发异常后，再决定采取 KeepChanges、KeepCurrentValues、OverwriteCurrentValues 等方式处理数据。

```
1 public class PersonDAL
2 {
3     delegate void MyDelegate(Person person);
4
5     static void Main(string[] args)
6     {
7         //在更新数据前显示对象信息
8         PersonDAL personDAL = new PersonDAL();
9         var beforeObj = personDAL.GetPerson(52);
10        personDAL.DisplayProperty("Before", beforeObj);
11
12        //更新Person的FirstName、SecondName属性
13        Person person1 = new Person();
14        person1.Id = 52;
15        person1.FirstName = "Mike";
16        person1.SecondName = "Wang";
17        person1.Age = 32;
18        person1.Address = "Tianhe";
19        person1.Telephone = "13660123456";
20        person1.Email = "Leslie@163.com";
21
22        //更新Person的FirstName、SecondName属性
23        Person person2 = new Person();
24        person2.Id = 52;
25        person2.FirstName = "Rose";
26        person2.SecondName = "Chen";
27        person2.Age = 32;
28        person2.Address = "Tianhe";
```

```
29     person2.Telephone = "13660123456";
30     person2.Email = "Leslie@163.com";
31
32     //使用异步方式更新数据
33     MyDelegate myDelegate = new MyDelegate(personDAL.Update);
34     myDelegate.BeginInvoke(person1, null, null);
35     myDelegate.BeginInvoke(person2, null, null);
36
37     //显示更新后的对象信息
38     Thread.Sleep(1000);
39     var afterObj = personDAL.GetPerson(52);
40     personDAL.DisplayProperty("After", afterObj);
41     Console.ReadKey();
42 }
43
44 public void Update(Person person)
45 {
46     using (BusinessDataContext context = new BusinessDataContext())
47     {
48         try
49         {
50             var person1 = context.Person.Where(x => x.Id == person.Id).First();
51             if (person1 != null)
52             {
53                 person1.Address = person.Address;
54                 person1.Age = person.Age;
55                 person1.Email = person.Email;
56                 person1.FirstName = person.FirstName;
57                 person1.SecondName = person.SecondName;
58                 person1.Telephone = person.Telephone;
59             }
60             //虚拟操作, 保证多个值同时提交
61             Thread.Sleep(100);
62             context.SubmitChanges(ConflictMode.ContinueOnConflict);
63             DisplayProperty("SubmitChanges Success", person);
64         }
65         catch (ChangeConflictException ex)
66         {
67             //保持最新输入的上下文数据
68             DisplayProperty("ChangeConflictException", person);
69             context.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
70             context.SubmitChanges();
71         }
72     }
73 }
74
75 public Person GetPerson(int id)
76 {
77     using (BusinessDataContext context = new BusinessDataContext())
78     {
79         var person = context.Person.Where(x => x.Id == id);
80         return person.First();
81     }
82 }
83
84 //显示对象相关属性
85 public void DisplayProperty(string message, Person person)
86 {
87     String data = string.Format("{0}\n Person Message:\n Id:{1} FirstName:{2} " +
88         "SecondName:{3} Age:{4}\n Address:{5} Telephone:{6} Email:{7}\n",
89         message, person.Id, person.FirstName, person.SecondName, person.Age,
90         person.Address, person.Telephone, person.Email);
91     Console.WriteLine(data);
92 }
93 }
```

例子当中使用 `RefreshMode.KeepChanges` 的方式处理并发, 系统会保存最新输入的数据。观察测试结果, 当系统发生第一次更新时, 数据成功保存到数据库当中。在 `DataContext` 未释放前, 再次输入数据, 引发了 `ChangeConflictException` 异常。在捕获此并发异常后, 系统以 `RefreshMode.KeepChanges` 方式进行处理, 最后新输入的数据成功保存到数据库当中。

[回到目录](#)

六、结合事务处理并发冲突

Entity Framework 中已经有比较完善的机制处理并发，但使用乐观性并发处理数据，一旦多个客户端同时更新同一张表格的同一个对象时，将会激发 `OptimisticConcurrencyException` 异常，系统必须预先定制好处理方案对此并发异常进行处理。结合事务处理并发异常，是一个比较高效的数据管理方式。事务能对数据的更新进行检测，一旦发现异常，便会实现回滚。本文会使用常用的隐式事务

`TransactionScope` 作为例子进行介绍，对事务的详细介绍，可以参考“[C#综合揭秘——细说事务](#)”。其实在CodePlex网站可以看到，微软在 `ObjectContext.SaveChanges` 方法的实现中，已经使用了事务用于保证数据输入的正确性。但对于一部分多表操作的方法中，依然需要使用事务在多表更新时保证一致性。

在实际开发过程当中，最常遇见的是在订单更新的过程，系统会把客户的个人信息，收货地址，联系方式，订单总体表，订单明细表等多表格的数据同时提交。此时，可以选择不同的方法进行处理。

在 `Order` 与 `OrderItem` 这种主从关系表中，可以使用把次表格原有的有关数据全部删除，然后根据新输入的对象重新加载的方法进行处理。此处理方法的好处在于，避免新输入的数据与次表格的原有数据存在冲突，但为此一旦发生并发就有可能引起多次加载的危险。

对于 `Person`，`Order` 这样不同的实体，一般会把操作放置于不同的操作对象中，以不同的 `ObjectContext` 进行处理。为了保证操作具有一致性，可使用事务把多个表格的操作合并在一起。此时，即使系统发生异常，多个表数据也能实现同步回滚，避免引起逻辑上的错误。

但使用事务必须注意，一旦发生并发，容易引发事务锁异常：“事务(进程 ID 53)与另一个进程被死锁在 锁 资源上，并且已被选作死锁牺牲品”。这是因为多个事务被同时触发，在实现更新的过程中事务都对部分数据实现锁定而引起。所以使用事务时需要引入锁来避免此情况出来。

```
1 public class OrderDAL
2 {
3     public void UpdateOrder(Order order)
4     {
5         using (BusinessEntities context = new BusinessEntities())
6         {
7             Order objResult = context.Order.Include("OrderItem")
8                 .Where(x => x.Id == order.Id).First();
9
10            if (objResult != null)
11            {
12                try
13                {
14                    //把原有相关的OrderItem对象全部删除
15                    foreach (var item in objResult.OrderItem.ToList())
16                        context.OrderItem.DeleteObject(item);
17                    //更新 Order对象的属性，加入对应的OrderItem对象
18                    context.ApplyCurrentValues("BusinessEntities.Order", order);
19                    foreach (var item in order.OrderItem.ToList())
20                    {
21                        //先把OrderItem的导航属性Order变为空值
22                        //否则系统会显示重复插入的异常
23                        item.Order = null;
24                        context.OrderItem.AddObject(item);
25                    }
26                    context.SaveChanges();
27                }
28                catch (System.Data.OptimisticConcurrencyException ex)
29                {
30                    { ..... }
31                }
32                catch (Exception ex)
33                {
34                    { ..... }
35                }
36            }
37        }
38    }
39
40    public class PersonDAL
41    {
42        //更新Person对象
43        public void Update(Person person)
44        {
45            using (BusinessEntities context = new BusinessEntities())
46            {
47                var obj = context.Person.Where(x => x.Id == person.Id).First();
```

```
45         if (obj != null)
46             context.ApplyCurrentValues("Person", person);
47         context.SaveChanges();
48     }
49 }
50 }
51
52 public class UpdateOrder
53 {
54     private OrderDAL orderDAL = new OrderDAL();
55     private PersonDAL personDAL = new PersonDAL();
56
57     public void DoWork(Person person, Order order)
58     {
59         lock(this)
60         {
61             //操作时将同时更新Order、OrderItem、Person三个表格
62             //为了避免其中一个操作出现错误而引起数据逻辑性错误
63             //应使用事务作为保护，在出现错误时实现同步回滚
64             using (TransactionScope scope = new TransactionScope())
65             {
66                 orderDAL.UpdateOrder(order);
67                 personDAL.Update(person);
68                 scope.Complete();
69             }
70         }
71     }
72 }
```

在复杂的多表数据处理过程中，推荐使用事务结合锁来进行处理。

值得注意的是：使用乐观并发与悲观并发方式最大区别在于，传统的悲观并发处理方式不允许同一时刻有多个客户端处理同一个数据表中的相同对象，但因为客观因素的影响，系统难以仔细辨认客户同时进行修改的是否同一个数据项，所以基本的做法是使用锁把更新对象进行锁定。但如此一来，无论客户同时更新的是否同一个数据项，操作都将会被延时或禁止。换句话说，无论需要更新的是相同对象还是不同对象，客户端都不能同时更新同一个数据表。在处理复杂的多表数据过程中，可以考虑使用此处理方式。

若使用乐观并发方式，系统允许多个客户端同时处理同一个数据表。如果所处理的是数据表中的不同对象，操作可以顺利地进行而不会相互影响。如果所处理的是数据表中的相同对象，可以保守处理，保存第一次输入的对象值，然后把引起异常的数据显示让客户进行对比。

[返回目录](#)

总结

并发话题与线程、进程等其他话题有所不同，它并没有复杂的类和方法。处理并发来来去去都是简单的几行代码，它所重视的是并发异常发生后所带来的后果与处理方式。与中国传统的太极相似，并发只重其意，不重其招，只要深入地了解其过程，考虑其可能带来的问题后，你便可以对其收发自如。

对并发的处理应该针对特定的问题，分别对待。到最后你可能发现，原来微软早已为你定制好处理的方式，可能“回到原点”什么都不做就是最好的处理方式。

希望此文章对大家有所帮助，欢迎各位提供宝贵意见，指出文中存在错误或漏洞。

对 .NET 开发有兴趣的朋友欢迎加入QQ群：[230564952](#) 共同探讨！

C#综合揭秘

- 通过修改注册表建立Windows自定义协议
- Entity Framework 并发处理详解
- 细说进程、应用程序域与上下文
- 细说多线程（上）
- 细说多线程（下）
- 细说事务
- 深入分析委托与事件

作者：风尘浪子

<http://www.cnblogs.com/leslies2/archive/2012/07/30/2608784.html>

分类：C#综合揭秘

原创作品，转载时请注明作者及出处

标签：C#并发处理, Entity Framewrok并发, 乐观并发, 悲观并发, LINQ to SQL并发, ConcurrencyMode, OptimisticConcurrencyException异常, ChangeConflictException异常, ConflictMode, 事务与并发

好文要顶

关注我

收藏该文



风尘浪子
关注 - 11
粉丝 - 1505
荣誉：推荐博客
[+加关注](#)

290

(请您对文章做出评价)

« 上一篇: [Apache2.2+Tomcat7.0整合配置详解](#)
» 下一篇: [Java远程通信技术——Axis实战](#)

posted on 2012-07-30 13:15 风尘浪子 阅读(36269) 评论(52) 编辑 收藏

评论

#51楼 2015-05-29 17:12 白细胞

@ 风尘浪子
我又看了下，不至于呀,我是DBfirst模式的，EDMX模型开发。该引用的什么system.data.objects和entity都有引用，但就是点不出来。
支持(0) 反对(0)

#52楼 2015-05-29 21:19 魑魅魍魉

@ 风尘浪子
非常感谢博主花时间给写demo，我现在明白应该如何处理这种情况了。我开始思路是这样的
比如我update(object)，我最开始一门心思想把object的某个字段，在调用update之前就加减好，我这种思路的话肯定是错误的。
应该在update方法里面的lock锁里面相加减，这样就不会有问题了，谢谢
支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问](#)网站首页。

- 最新IT新闻：
- 甲骨文在Java问题上不再沉默
 - 微软称将为Linux用户带来“令人振奋的消息”
 - Eclipse基金会更新了其使用条款和隐私政策
 - 火星上神秘的暗条纹意味着流动水的存在
 - 关于苹果产品的8大传闻：从iPhone 7 Pro到智能音箱
- » 更多新闻...

- 最新知识库文章：
- 写给初学前端工程师的一封信
 - 抽象：程序员必备的能力
 - 编程同写作，写代码只是在码字
 - 遇见程序员男友
 - 设计师的视觉设计五项修炼
- » 更多知识库文章...