LINQ to SQL语句(1)之Where

适用场景：实现过滤，查询等功能。

说明：与SQL命令中的Where作用相似，都是起到范围限定也就是过滤作用的，而判断条件就是它后面所接的子句。Where操作包括3种形式，分别为简单形式、关系条件形式、First()形式。下面分别用实例举例下：

1. 简单形式：

例如：使用where筛选在伦敦的客户

```
var q =
    from c in db.Customers
    where c.City == "London"
    select c;
```

再如：筛选1994 年或之后雇用的雇员：

```
var q =
    from e in db.Employees
    where e.HireDate >= new DateTime(1994, 1, 1)
    select e;
```

2. 关系条件形式：

筛选库存量在订货点水平之下但未断货的产品：

```
var q =
    from p in db.Products
    where p.UnitsInStock <= p.ReorderLevel && !p.Discontinued
    select p;
var q =
    from p in db.Products
    where p.UnitPrice > 10m || p.Discontinued
    select p;
```

下面这个例子是调用两次where以筛选出UnitPrice大于10且已停产的产品。

```
var q =
    db.Products.Where(p=>p.UnitPrice > 10m).Where(p=>p.Discontinued);
```

3. First()形式：

返回集合中的一个元素，其实质就是在SQL语句中加TOP (1)。

简单用法：选择表中的第一个发货方。

```
Shipper shipper = db.Shippers.First();
```

元素：选择CustomerID 为“BONAP”的单个客户

```
Customer cust = db.Customers.First(c => c.CustomerID == "BONAP");
```

条件：选择运费大于 10.00 的订单：

```
Order ord = db.Orders.First(o => o.Freight > 10.00M);
```

LINQ to SQL语句(2)之Select/Distinct

适用场景：o(∩_∩)o... 查询呗。

说明：和SQL命令中的select作用相似但位置不同，查询表达式中的select及所接子句是放在表达式最后并把子句中的变量也就是结果返回回来；延迟。Select/Distinct操作包括9种形式，分别为简单用法、匿名类型形式、条件形式、指定类型形式、筛选形式、整形类型形式、嵌套类型形式、本地方法调用形式、Distinct形式。

1. 简单用法：

这个示例返回仅含客户联系人姓名的序列。

```
var q =
    from c in db.Customers
    select c.ContactName;
```

注意：这个语句只是一个声明或者一个描述，并没有真正把数据取出来，只有当你需要该数据的时候，它才会执行这个语句，这就是延迟加载(deferred loading)。如果，在声明的时候就返回的结果集是对象的集合。你可以使用ToList() 或ToArray()方法把查询结果先进行保存，然后再对这个集合进行查询。当然延迟加载(deferred loading)可以像拼接SQL语句那样拼接查询语法，再执行它。

2. 匿名类型形式：

说明：匿名类型是C#3.0中新特性。其实质是编译器根据我们自定义自动产生一个匿名的类来帮助我们实现临时变量的储存。匿名类型还依赖于另外一个特性：支持根据property来创建对象。比如，var d = new { Name = "s" };编译器自动产生一个有property叫做Name的匿名类，然后按这个类型分配内存，并初始化对象。但是var d = new {"s"};是编译不通过的。因为，编译器不知道匿名类中的property的名字。例如string c = "d";var d = new { c}; 则是可以通过编译的。编译器会创建一个叫做匿名类带有叫c的property。 例如下例：new{c,ContactName,c.Phone};ContactName和Phone都是在映射文件中定义与表中字段相对应的property。编译器读取数据并创建对象时，会创建一个匿名类，这个类有两个属性，为ContactName和Phone，然后根据数据初始化对象。另外编译器还可以重命名property的名字。

```
var q =
    from c in db.Customers
    select new {c.ContactName, c.Phone};
```

上面语句描述：使用 SELECT 和匿名类型返回仅含客户联系人姓名和电话号码的序列

```
var q =
    from e in db.Employees
    select new
    {
        Name = e.FirstName + " " + e.LastName,
```

```
        Phone = e.HomePhone
    };
```

上面语句描述：使用SELECT和匿名类型返回仅含雇员姓名和电话号码的序列，并将FirstName和LastName字段合并为一个字段“Name”，此外在所得的序列中将HomePhone字段重命名为Phone。

```
var q =
    from p in db.Products
    select new
    {
        p.ProductID,
        HalfPrice = p.UnitPrice / 2
    };
```

上面语句描述：使用SELECT和匿名类型返回所有产品的ID以及HalfPrice(设置为产品单价除以2所得的值)的序列。

3. 条件形式：

说明：生成SQL语句为：case when condition then else。

```
var q =
    from p in db.Products
    select new
    {
        p.ProductName,
        Availability =
        p.UnitsInStock - p.UnitsOnOrder < 0 ?
        "Out Of Stock" : "In Stock"
    };
```

上面语句描述：使用SELECT和条件语句返回产品名称和产品供货状态的序列。

4. 指定类型形式：

说明：该形式返回你自定义类型的对象集。

```
var q =
    from e in db.Employees
    select new Name
    {
        FirstName = e.FirstName,
        LastName = e.LastName
    };
```

上面语句描述：使用SELECT和已知类型返回雇员姓名的序列。

5. 筛选形式：

说明：结合where使用，起到过滤作用。

```
var q =
    from c in db.Customers
    where c.City == "London"
    select c.ContactName;
```

上面语句描述：使用SELECT和WHERE返回仅含伦敦客户联系人姓名的序列。

6. shaped形式（整形类型）：

说明：其select操作使用了匿名对象，而这个匿名对象中，其属性也是个匿名对象。

```
var q =
    from c in db.Customers
    select new {
        c.CustomerID,
        CompanyInfo = new {c.CompanyName, c.City, c.Country},
        ContactInfo = new {c.ContactName, c.ContactTitle}
    };
```

语句描述：使用SELECT 和匿名类型返回有关客户的数据的整形子集。查询顾客的ID和公司信息（公司名称，城市，国家）以及联系信息（联系人和职位）。

7. 嵌套类型形式：

说明：返回的对象集中的每个对象DiscountedProducts属性中，又包含一个集合。也就是每个对象也是一个集合类。

```
var q =
    from o in db.Orders
    select new {
        o.OrderID,
        DiscountedProducts =
            from od in o.OrderDetails
            where od.Discount > 0.0
            select od,
        FreeShippingDiscount = o.Freight
    };
```

语句描述：使用嵌套查询返回所有订单及其OrderID 的序列、打折订单中项目的子序列以及免送货所省下的金额。

8. 本地方法调用形式（LocalMethodCall）：

这个例子在查询中调用本地方法PhoneNumberConverter将电话号码转换为国际格式。

```
var q = from c in db.Customers
         where c.Country == "UK" || c.Country == "USA"
         select new
         {
             c.CustomerID,
             c.CompanyName,
```

```
            Phone = c.Phone,
            InternationalPhone =
            PhoneNumberConverter(c.Country, c.Phone)
        };
```

PhoneNumberConverter方法如下：

```
public string PhoneNumberConverter(string Country, string Phone)
{
    Phone = Phone.Replace(" ", "").Replace(")", ")-");
    switch (Country)
    {
        case "USA":
            return "1-" + Phone;
        case "UK":
            return "44-" + Phone;
        default:
            return Phone;
    }
}
```

下面也是使用了这个方法将电话号码转换为国际格式并创建XDocument

```
XDocument doc = new XDocument(
    new XElement("Customers", from c in db.Customers
              where c.Country == "UK" || c.Country == "USA"
              select (new XElement("Customer",
                      new XAttribute("CustomerID", c.CustomerID),
                      new XAttribute("CompanyName", c.CompanyName),
                      new XAttribute("InterationalPhone",
                       PhoneNumberConverter(c.Country, c.Phone))
                     ))));
```

9.Distinct形式：

说明：筛选字段中不相同的值。用于查询不重复的结果集。生成SQL语句为：SELECT DISTINCT [City] FROM [Customers]

```
var q = (
    from c in db.Customers
    select c.City )
    .Distinct();
```

语句描述：查询顾客覆盖的国家。

LINQ to SQL语句(3)之Count/Sum/Min/Max/Avg

适用场景：统计数据吧，比如统计一些数据的个数，求和，最小值，最大值，平均数。

Count

说明：返回集合中的元素个数，返回INT类型；不延迟。生成SQL语句为：SELECT COUNT(*) FROM

1.简单形式：

得到数据库中客户的数量：

```
var q = db.Customers.Count();
```

2. 带条件形式：

得到数据库中未断货产品的数量：

```
var q = db.Products.Count(p => !p.Discontinued);
```

LongCount

说明：返回集合中的元素个数，返回LONG类型；不延迟。对于元素个数较多的集合可视情况可以选用LongCount来统计元素个数，它返回long类型，比较精确。生成SQL语句为：SELECT COUNT_BIG(*) FROM

```
var q = db.Customers.LongCount();
```

Sum

说明：返回集合中数值类型元素之和，集合应为INT类型集合；不延迟。生成SQL语句为：SELECT SUM(…) FROM

1. 简单形式：

得到所有订单的总运费：

```
var q = db.Orders.Select(o => o.Freight).Sum();
```

2.映射形式：

得到所有产品的订货总数：

```
var q = db.Products.Sum(p => p.UnitsOnOrder);
```

Min

说明：返回集合中元素的最小值；不延迟。生成SQL语句为：SELECT MIN(…) FROM

1. 简单形式：

查找任意产品的最低单价：

```
var q = db.Products.Select(p => p.UnitPrice).Min();
```

2. 映射形式：

查找任意订单的最低运费：

```
var q = db.Orders.Min(o => o.Freight);
```

3.元素：

查找每个类别中单价最低的产品：

```
var categories =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        CategoryID = g.Key,
```

```
        CheapestProducts =
            from p2 in g
            where p2.UnitPrice == g.Min(p3 => p3.UnitPrice)
            select p2
    };
```

Max

说明：返回集合中元素的最大值；不延迟。生成SQL语句为：SELECT MAX(...) FROM

1. 简单形式：

查找任意雇员的最近雇用日期：

```
var q = db.Employees.Select(e => e.HireDate).Max();
```

2. 映射形式：

```
var q = db.Products.Max(p => p.UnitsInStock);
```

3. 元素：

查找每个类别中单价最高的产品：

```
var categories =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        MostExpensiveProducts =
            from p2 in g
            where p2.UnitPrice == g.Max(p3 => p3.UnitPrice)
            select p2
    };
```

Average

说明：返回集合中的数值类型元素的平均值。集合应为数字类型集合，其返回值类型为double；不延迟。生成SQL语句为：SELECT AVG(...) FROM

1. 简单形式：

得到所有订单的平均运费：

```
var q = db.Orders.Select(o => o.Freight).Average();
```

2. 映射形式：

得到所有产品的平均单价：

```
var q = db.Products.Average(p => p.UnitPrice);
```

3. 元素：

查找每个类别中单价高于该类别平均单价的产品：

```
var categories =
    from p in db.Products
    group p by p.CategoryID into g
```

```
    select new {
        g.Key,
        ExpensiveProducts =
            from p2 in g
            where p2.UnitPrice > g.Average(p3 => p3.UnitPrice)
            select p2
    };
```

Aggregate

说明：根据输入的表达式获取聚合值；不延迟。即是说：用一个种子值与当前元素通过指定的函数来进行对比来遍历集合中的元素，符合条件的元素保留下来。如果没有指定种子值的话，种子值默认为集合的第一个元素。

LINQ to SQL语句(4)之Join

适用场景：在我们表关系中有一对一关系，一对多关系，多对多关系等。对各个表之间的关系，就用这些实现对多个表的操作。

说明：在Join操作中，分别为Join(Join查询), SelectMany(Select一对多选择)和GroupJoin(分组Join查询)。 该扩展方法对两个序列中键匹配的元素进行inner join操作

SelectMany

说明：我们在写查询语句时，如果被翻译成SelectMany需要满足2个条件。1：查询语句中没有join和into，2：必须出现EntitySet。在我们表关系中有一对一关系，一对多关系，多对多关系等，下面分别介绍一下。

1.一对多关系(1 to Many)：

```
var q =
    from c in db.Customers
    from o in c.Orders
    where c.City == "London"
    select o;
```

语句描述：Customers与Orders是一对多关系。即Orders在Customers类中以EntitySet形式出现。所以第二个from是从c.Orders而不是db.Orders里进行筛选。这个例子在From子句中使用外键导航选择伦敦客户的所有订单。

```
var q =
    from p in db.Products
    where p.Supplier.Country == "USA" && p.UnitsInStock == 0
    select p;
```

语句描述：这一句使用了p.Supplier.Country条件，间接关联了Supplier表。这个例子在Where子句中使用外键导航筛选其供应商在美国且缺货的产品。生成SQL语句为：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0].[CategoryID],[t0].[QuantityPerUnit],[t0].[UnitPrice],
```

```
[t0].[UnitsInStock], [t0].[UnitsOnOrder],[t0].[ReorderLevel],
[t0].[Discontinued] FROM [dbo].[Products] AS [t0]
LEFT OUTER JOIN [dbo].[Suppliers] AS [t1] ON
[t1].[SupplierID] = [t0].[SupplierID]
WHERE ([t1].[Country] = @p0) AND ([t0].[UnitsInStock] = @p1)
-- @p0: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p1: Input Int (Size = 0; Prec = 0; Scale = 0) [0]
```

2. 多对多关系(Many to Many)：

```
var q =
    from e in db.Employees
    from et in e.EmployeeTerritories
    where e.City == "Seattle"
    select new
    {
        e.FirstName,
        e.LastName,
        et.Territory.TerritoryDescription
    };
```

说明：多对多关系一般会涉及三个表(如果有一个表是自关联的，那有可能只有2个表)。这一句语句涉及Employees, EmployeeTerritories, Territories三个表。它们的关系是1：M：1。Employees和Territories没有很明确的关系。

语句描述：这个例子在From子句中使用外键导航筛选在西雅图的雇员，同时列出其所在地区。这条生成SQL语句为：

```
SELECT [t0].[FirstName], [t0].[LastName], [t2].[TerritoryDescription]
FROM [dbo].[Employees] AS [t0] CROSS JOIN [dbo].[EmployeeTerritories]
AS [t1] INNER JOIN [dbo].[Territories] AS [t2] ON
[t2].[TerritoryID] = [t1].[TerritoryID]
WHERE ([t0].[City] = @p0) AND ([t1].[EmployeeID] = [t0].[EmployeeID])
-- @p0: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [Seattle]
```

3. 自联接关系：

```
var q =
    from e1 in db.Employees
    from e2 in e1.Employees
    where e1.City == e2.City
    select new {
        FirstName1 = e1.FirstName, LastName1 = e1.LastName,
        FirstName2 = e2.FirstName, LastName2 = e2.LastName,
        e1.City
    };
```

语句描述：这个例子在select 子句中使用外键导航筛选成对的雇员，每对中一个雇员隶属于另一个雇员，且两个雇员都来自相同城市。生成SQL语句为：

```
SELECT [t0].[FirstName] AS [FirstName1], [t0].[LastName] AS
[LastName1],[t1].[FirstName] AS [FirstName2], [t1].[LastName] AS
[LastName2],[t0].[City] FROM [dbo].[Employees] AS [t0],
[dbo].[Employees] AS [t1] WHERE ([t0].[City] = [t1].[City]) AND
([t1].[ReportsTo] = [t0].[EmployeeID])
```

GroupJoin

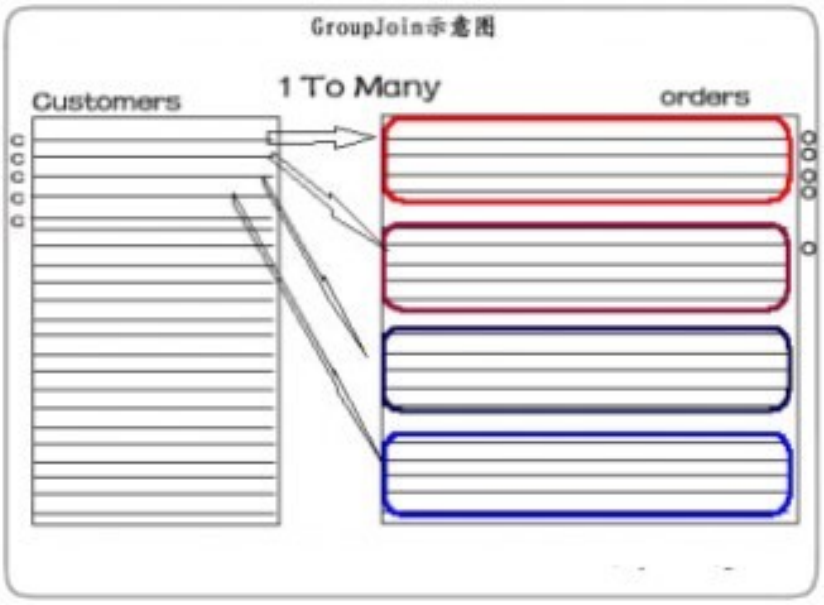
像上面所说的，没有join和into，被翻译成SelectMany，同时有join和into时，那么就被翻译为GroupJoin。在这里into的概念是对其结果进行重新命名。

1. 双向联接(Two way join)：

此示例显式联接两个表并从这两个表投影出结果：

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID
    equals o.CustomerID into orders
    select new
    {
        c.ContactName,
        OrderCount = orders.Count()
    };
```

说明：在一对多关系中，左边是1，它每条记录为c（from c in db.Customers），右边是Many，其每条记录叫做o（join o in db.Orders），每对应左边的一个c，就会有一组o，那这一组o，就叫做orders，也就是说，我们把一组o命名为orders，这就是into用途。这也就是为什么在select语句中，orders可以调用聚合函数Count。在T-SQL中，使用其内嵌的T-SQL返回值作为字段值。如图所示：



生成SQL语句为：

```
SELECT [t0].[ContactName], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t1]
    WHERE [t0].[CustomerID] = [t1].[CustomerID]
) AS [OrderCount]
FROM [dbo].[Customers] AS [t0]
```

2.三向联接(There way join)：

此示例显式联接三个表并分别从每个表投影出结果：

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID
    equals o.CustomerID into ords
    join e in db.Employees on c.City
    equals e.City into emps
    select new
    {
        c.ContactName,
        ords = ords.Count(),
        emps = emps.Count()
    };
```

生成SQL语句为：

```
SELECT [t0].[ContactName], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t1]
    WHERE [t0].[CustomerID] = [t1].[CustomerID]
) AS [ords], (
```

```
SELECT COUNT(*)
    FROM [dbo].[Employees] AS [t2]
    WHERE [t0].[City] = [t2].[City]
) AS [emps]
FROM [dbo].[Customers] AS [t0]
```

3.左外部联接(Left Outer Join):

此示例说明如何通过使用 此示例说明如何通过使用DefaultIfEmpty() 获取左外部联接。在雇员没有订单时，DefaultIfEmpty()方法返回null:

```
var q =
    from e in db.Employees
    join o in db.Orders on e equals o.Employee into ords
    from o in ords.DefaultIfEmpty()
    select new
    {
        e.FirstName,
        e.LastName,
        Order = o
    };
```

说明: 以Employees左表，Orders右表，Orders 表中为空时，用null值填充。Join的结果重命名ords，使用DefaultIfEmpty()函数对其再次查询。其最后的结果中有个Order，因为from o in ords.DefaultIfEmpty() 是对ords组再一次遍历，所以，最后结果中的Order并不是一个集合。但是，如果没有from o in ords.DefaultIfEmpty() 这句，最后的select语句写成select new { e.FirstName, e.LastName, Order = ords }的话，那么Order就是一个集合。

4. 投影的Let赋值(Projected let assignment):

说明: let语句是重命名。let位于第一个from和select语句之间。

这个例子从联接投影出最终"Let"表达式:

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID
    equals o.CustomerID into ords
    let z = c.City + c.Country
    from o in ords
    select new
    {
        c.ContactName,
        o.OrderID,
        z
    };
```

5.组合键(Composite Key):

这个例子显示带有组合键的联接:

```
var q =
    from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new
        {
            o.OrderID,
            p.ProductID
        } equals
            new
            {
                d.OrderID,
                d.ProductID
            }
        into details
    from d in details
    select new
    {
        o.OrderID,
        p.ProductID,
        d.UnitPrice
    };
```

说明: 使用三个表，并且用匿名类来说明：使用三个表，并且用匿名类来表示它们之间的关系。它们之间的关系不能用一个键描述清楚，所以用匿名类，来表示组合键。还有一种是两个表之间是用组合键表示关系的，不需要使用匿名类。

6. 可为null/不可为null的键关系(Nullable/Nonnullable Key Relationship):

这个实例显示如何构造一侧可为 null 而另一侧不可为 null 的联接:

```
var q =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals
        (int?)e.EmployeeID into emps
    from e in emps
    select new
    {
        o.OrderID,
```

```
        e.FirstName
    };
```

LINQ to SQL语句(5)之Order By

适用场景：对查询出的语句进行排序，比如按时间排序等等。

说明：按指定表达式对集合排序；延迟，：按指定表达式对集合排序；延迟，默认是升序，加上descending表示降序，对应的扩展方法是OrderBy和OrderByDescending

1. 简单形式

这个例子使用 orderby 按雇用日期对雇员进行排序：

```
var q =
    from e in db.Employees
    orderby e.HireDate
    select e;
```

说明：默认为升序

2. 带条件形式

注意：Where和Order By的顺序并不重要。而在T-SQL中，Where和Order By有严格的位置限制。

```
var q =
    from o in db.Orders
    where o.ShipCity == "London"
    orderby o.Freight
    select o;
```

语句描述：使用where和orderby按运费进行排序。

3. 降序排序

```
var q =
    from p in db.Products
    orderby p.UnitPrice descending
    select p;
```

4.ThenBy

语句描述：使用复合的 orderby 对客户进行排序，进行排序：

```
var q =
    from c in db.Customers
    orderby c.City, c.ContactName
    select c;
```

说明：按多个表达式进行排序，例如先按City排序，当City相同时，按ContactName排序。这一句用Lambda表达式像这样写：

```
var q =
    .OrderBy(c => c.City)
    .ThenBy(c => c.ContactName).ToList();
```

在T-SQL中没有ThenBy语句，其依然翻译为OrderBy，所以也可以用下面语句来表达：

```
var q =
    db.Customers
    .OrderBy(c => c.ContactName)
    .OrderBy(c => c.City).ToList();
```

所要注意的是，多个OrderBy操作时，级连方式是按逆序。对于降序的，用相应的降序操作符替换即可。

```
var q =
    db.Customers
    .OrderByDescending(c => c.City)
    .ThenByDescending(c => c.ContactName).ToList();
```

需要说明的是，OrderBy操作，不支持按type排序，也不支持匿名类。比如

```
var q =
    db.Customers
    .OrderBy(c => new
    {
        c.City,
        c.ContactName
    }).ToList();
```

会被抛出异常。错误是前面的操作有匿名类，再跟OrderBy时，比较的是类别。比如

```
var q =
    db.Customers
    .Select(c => new
    {
        c.City,
        c.Address
    })
    .OrderBy(c => c).ToList();
```

如果你想使用OrderBy(c => c)，其前提条件是，前面步骤中，所产生的对象的类别必须为C#语言的基本类型。比如下句，这里City为string类型。

```
var q =
    db.Customers
    .Select(c => c.City)
    .OrderBy(c => c).ToList();
```

5.ThenByDescending

这两个扩展方式都是用在OrderBy/OrderByDescending之后的，第一个ThenBy/ThenByDescending扩展方法作为第二位排序依据，第二个ThenBy/ThenByDescending则作为第三位排序依据，以此类推

```
var q =
    from o in db.Orders
    where o.EmployeeID == 1
    orderby o.ShipCountry, o.Freight descending
    select o;
```

语句描述：使用orderby先按发往国家再按运费从高到低的顺序对 EmployeeID 1 的订单进行排序。

6. 带GroupBy形式

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    orderby g.Key
    select new {
        g.Key,
        MostExpensiveProducts =
            from p2 in g
            where p2.UnitPrice == g.Max(p3 => p3.UnitPrice)
            select p2
    };
```

语句描述：使用orderby、Max 和 Group By 得出每种类别中单价最高的产品，并按 CategoryID 对这组产品进行排序。

LINQ to SQL语句(6)之Group By/Having

适用场景：分组数据，为我们查找数据缩小范围。

说明：分配并返回对传入参数进行分组操作后的可枚举对象。分组；延迟

1. 简单形式：

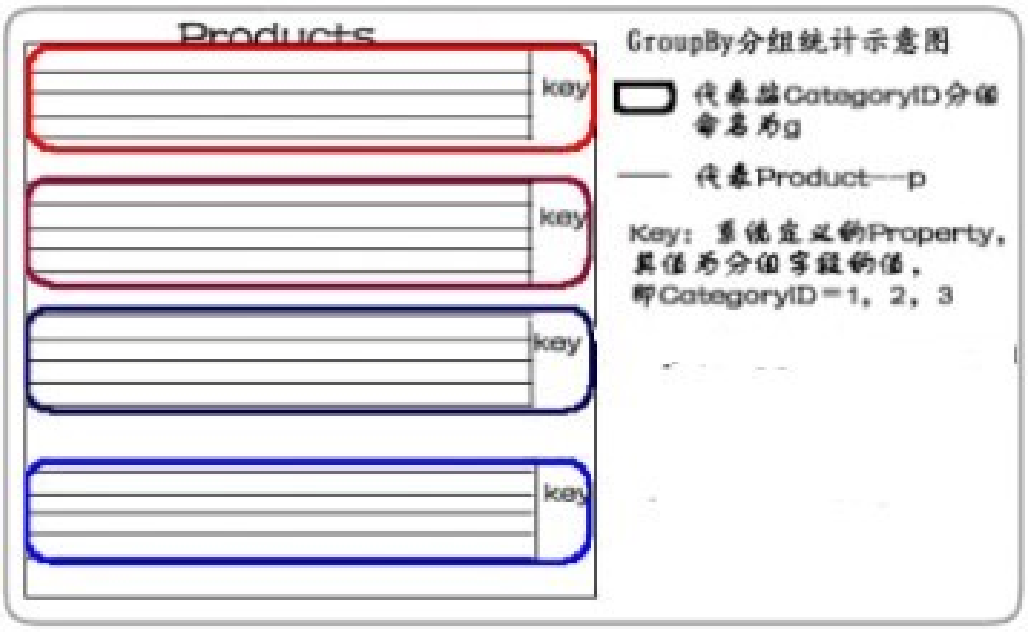
```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select g;
```

语句描述：使用Group By按CategoryID划分产品。

说明：from p in db.Products 表示从表中将产品对象取出来。group p by p.CategoryID into g表示对p按CategoryID字段归类。其结果命名为g，一旦重新命名，p的作用域就结束了，所以，最后select时，只能select g。当然，也不必重新命名可以这样写：

```
var q =
    from p in db.Products
    group p by p.CategoryID;
```

我们用示意图表示：



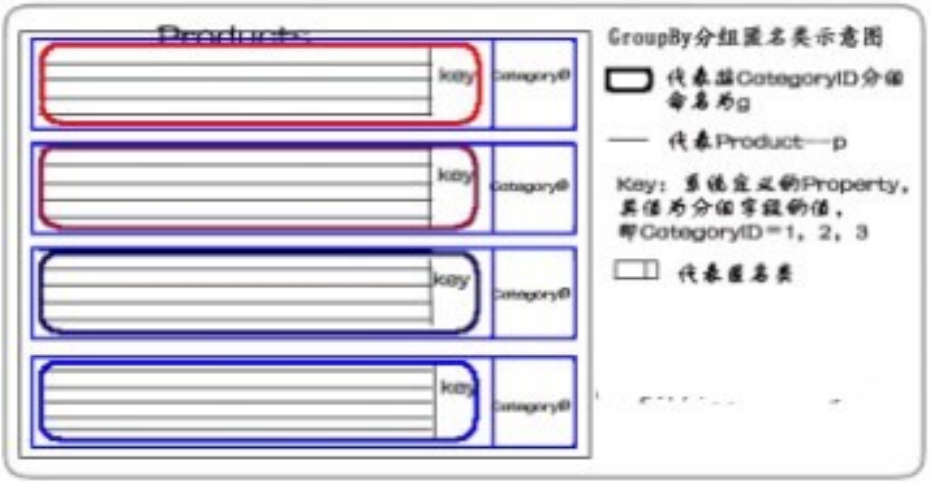
如果想遍历某类别中所有记录，这样：

```
foreach (var gp in q)
{
    if (gp.Key == 2)
    {
        foreach (var item in gp)
        {
            //do something
        }
    }
}
```

2.Select匿名类：

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new { CategoryID = g.Key, g };
```

说明：在这句LINQ语句中，有2个property：CategoryID和g。这个匿名类，其实质是对返回结果集重新进行了包装。把g的property封装成一个完整的分组。如下图所示：



如果想遍历某匿名类中所有记录，要这么做：

```
foreach (var gp in q)
{
    if (gp.CategoryID == 2)
    {
        foreach (var item in gp.g)
        {
            //do something
        }
    }
}
```

3.最大值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        MaxPrice = g.Max(p => p.UnitPrice)
    };
```

语句描述：使用Group By和Max查找每个CategoryID的最高单价。

说明：先按CategoryID归类，判断各个分类产品中单价最大的Products。取出CategoryID值，并把UnitPrice值赋给MaxPrice。

4. 最小值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
```

```
        MinPrice = g.Min(p => p.UnitPrice)
    };
```

语句描述：使用Group By和Min查找每个CategoryID的最低单价。

说明：先按CategoryID归类，判断各个分类产品中单价最小的Products。取出CategoryID值，并把UnitPrice值赋给MinPrice。

5. 平均值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        AveragePrice = g.Average(p => p.UnitPrice)
    };
```

语句描述：使用Group By和Average得到每个CategoryID的平均单价。

说明：先按CategoryID归类，取出CategoryID值和各个分类产品中单价的平均值。

6. 求和

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        TotalPrice = g.Sum(p => p.UnitPrice)
    };
```

语句描述：使用Group By和Sum得到每个CategoryID 的单价总计。

说明：先按CategoryID归类，取出CategoryID值和各个分类产品中单价的总和。

7. 计数

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        NumProducts = g.Count()
    };
```

语句描述：使用Group By和Count得到每个CategoryID中产品的数量。

说明：先按CategoryID归类，取出CategoryID值和各个分类产品的数量。

8. 带条件计数

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
```

```
    select new {
        g.Key,
        NumProducts = g.Count(p => p.Discontinued)
    };
```

语句描述：使用Group By和Count得到每个CategoryID中断货产品的数量。

说明：先按CategoryID归类，取出CategoryID值和各个分类产品的断货数量。 Count函数里，使用了Lambda表达式，Lambda表达式中的p，代表这个组里的一个元素或对象，即某一个产品。

9. Where限制

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    where g.Count() >= 10
    select new {
        g.Key,
        ProductCount = g.Count()
    };
```

语句描述：根据产品的一ID分组，查询产品数量大于10的ID和产品数量。这个示例在Group By子句后使用Where子句查找所有至少有10种产品的类别。

说明：在翻译成SQL语句时，在最外层嵌套了Where条件。

10. 多列(Multiple Columns)

```
var categories =
    from p in db.Products
    group p by new
    {
        p.CategoryID,
        p.SupplierID
    }
        into g
        select new
            {
                g.Key,
                g
            };
```

语句描述：使用Group By按CategoryID和SupplierID将产品分组。

说明：既按产品的分类，又按供应商分类。在by后面，new出来一个匿名类。这里，Key其实质是一个类的对象，Key包含两个Property：CategoryID、SupplierID。用g.Key.CategoryID可以遍历CategoryID的值。

11. 表达式(Expression)

```
var categories =
    from p in db.Products
    group p by new { Criterion = p.UnitPrice > 10 } into g
    select g;
```

语句描述：使用Group By返回两个产品序列。第一个序列包含单价大于10的产品。第二个序列包含单价小于或等于10的产品。

说明：按产品单价是否大于10分类。其结果分为两类，大于的是一类，小于及等于为另一类。

LINQ to SQL语句(7)之Exists/In/Any/All/Contains

适用场景：用于判断集合中元素，进一步缩小范围。

Any

说明：用于判断集合中是否有元素满足某一条件；不延迟。（若条件为空，则集合只要不为空就返回True，否则为False）。有2种形式，分别为简单形式和带条件形式。

1. 简单形式：

仅返回没有订单的客户：

```
var q =
    from c in db.Customers
    where !c.Orders.Any()
    select c;
```

生成SQL语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE NOT (EXISTS(
    SELECT NULL AS [EMPTY] FROM [dbo].[Orders] AS [t1]
    WHERE [t1].[CustomerID] = [t0].[CustomerID]
   ))
```

2. 带条件形式：

仅返回至少有一种产品断货的类别：

```
var q =
    from c in db.Categories
    where c.Products.Any(p => p.Discontinued)
    select c;
```

生成SQL语句为：

```
SELECT [t0].[CategoryID], [t0].[CategoryName], [t0].[Description],
[t0].[Picture] FROM [dbo].[Categories] AS [t0]
```

```
WHERE EXISTS(
    SELECT NULL AS [EMPTY] FROM [dbo].[Products] AS [t1]
    WHERE ([t1].[Discontinued] = 1) AND
    ([t1].[CategoryID] = [t0].[CategoryID])
    )
```

All

说明：用于判断集合中所有元素是否都满足某一条件；不延迟

1. 带条件形式

```
var q =
    from c in db.Customers
    where c.Orders.All(o => o.ShipCity == c.City)
    select c;
```

语句描述：这个例子返回所有订单都运往其所在城市的客户或未下订单的客户。

Contains

说明：用于判断集合中是否包含有某一元素；不延迟。它是对两个序列进行连接操作的。

```
string[] customerID_Set =
    new string[] { "AROUT", "BOLID", "FISSA" };
var q = (
    from o in db.Orders
    where customerID_Set.Contains(o.CustomerID)
    select o).ToList();
```

语句描述：查找"AROUT", "BOLID" 和 "FISSA" 这三个客户的订单。先定义了一个数组，在LINQ to SQL中使用Contains，数组中包含了所有的CustomerID，即返回结果中，所有的CustomerID都在这个集合内。也就是in。 你也可以把数组的定义放在LINQ to SQL语句里。比如：

```
var q = (
    from o in db.Orders
    where (
    new string[] { "AROUT", "BOLID", "FISSA" })
    .Contains(o.CustomerID)
    select o).ToList();
```

Not Contains则取反：

```
var q = (
    from o in db.Orders
    where !(
    new string[] { "AROUT", "BOLID", "FISSA" })
    .Contains(o.CustomerID)
    select o).ToList();
```

1.包含一个对象：

```
var order = (from o in db.Orders
             where o.OrderID == 10248
             select o).First();
var q = db.Customers.Where(p => p.Orders.Contains(order)).ToList();
foreach (var cust in q)
{
    foreach (var ord in cust.Orders)
    {
        //do something
    }
}
```

语句描述：这个例子使用Contain查找哪个客户包含OrderID为10248的订单。

2. 包含多个值：

```
string[] cities =
    new string[] { "Seattle", "London", "Vancouver", "Paris" };
var q = db.Customers.Where(p=>cities.Contains(p.City)).ToList();
```

语句描述：这个例子使用Contains查找其所在城市为西雅图、伦敦、巴黎或温哥华的客户。

LINQ to SQL语句(8)之Concat/Union/Intersect/Except

适用场景：对两个集合的处理，例如追加、合并、取相同项、相交项等等。

Concat（连接）

说明：连接不同的集合，不会自动过滤相同项；延迟。

1. 简单形式：

```
var q = (
         from c in db.Customers
         select c.Phone
        ).Concat(
         from c in db.Customers
         select c.Fax
        ).Concat(
         from e in db.Employees
         select e.HomePhone
        );
```

语句描述：返回所有消费者和雇员的电话和传真。

2. 复合形式：

```
var q = (
         from c in db.Customers
         select new
         {
```

```
            Name = c.CompanyName,
            c.Phone
        }
    ).Concat(
        from e in db.Employees
        select new
        {
            Name = e.FirstName + " " + e.LastName,
            Phone = e.HomePhone
        }
    );
```

语句描述：返回所有消费者和雇员的姓名和电话。

Union（合并）

说明：连接不同的集合，自动过滤相同项；延迟。即是将两个集合进行合并操作，过滤相同的项。

```
var q = (
        from c in db.Customers
        select c.Country
    ).Union(
        from e in db.Employees
        select e.Country
    );
```

语句描述：查询顾客和职员所在的国家。

Intersect（相交）

说明：取相交项；延迟。即是获取不同集合的相同项（交集）。即先遍历第一个集合，找出所有唯一的元素，然后遍历第二个集合，并将每个元素与前面找出的元素作对比，返回所有在两个集合内都出现的元素。

```
var q = (
        from c in db.Customers
        select c.Country
    ).Intersect(
        from e in db.Employees
        select e.Country
    );
```

语句描述：查询顾客和职员同在的国家。

Except（与非）

说明：排除相交项；延迟。即是从某集合中删除与另一个集合中相同的项。先遍历第一个集合，找出所有唯一的元素，然后再遍历第二个集合，返回第二个集合中所有未出现在前面所

得元素集合中的元素。

```
var q = (
         from c in db.Customers
         select c.Country
        ).Except(
         from e in db.Employees
         select e.Country
        );
```

语句描述：查询顾客和职员不同的国家。

LINQ to SQL语句(9)之Top/Bottom和Paging和SqlMethods

适用场景：适量的取出自己想要的数据，不是全部取出，这样性能有所加强。

Take

说明：获取集合的前n个元素；延迟。即只返回限定数量的结果集。

```
var q = (
    from e in db.Employees
    orderby e.HireDate
    select e)
    .Take(5);
```

语句描述：选择所雇用的前5个雇员。

Skip

说明：跳过集合的前n个元素；延迟。即我们跳过给定的数目返回后面的结果集。

```
var q = (
    from p in db.Products
    orderby p.UnitPrice descending
    select p)
    .Skip(10);
```

语句描述：选择10种最贵产品之外的所有产品。

TakeWhile

SkipWhile

说明：直到某一条件成立就停止跳过；延迟。即用其条件去判断源序列中的元素并且跳过第一个符合判断条件的元素，一旦判断返回false，接下来将不再进行判断并返回剩下的所有元素。

Paging（分页）操作

适用场景：结合Skip和Take就可实现对数据分页操作。

1.索引

```
var q = (
    from c in db.Customers
    orderby c.ContactName
```

```
    select c)
    .Skip(50)
    .Take(10);
```

语句描述：使用Skip和Take运算符进行分页，跳过前50条记录，然后返回接下来10条记录，因此提供显示Products表第6页的数据。

2. 按唯一键排序

```
var q = (
    from p in db.Products
    where p.ProductID > 50
    orderby p.ProductID
    select p)
    .Take(10);
```

语句描述：使用Where子句和Take运算符进行分页，首先筛选得到仅50 (第5页最后一个ProductID)以上的ProductID，然后按ProductID排序，最后取前10个结果，因此提供Products表第6页的数据。请注意，此方法仅适用于按唯一键排序的情况。

SqlMethods操作

在LINQ to SQL语句中，为我们提供了SqlMethods操作，进一步为我们提供了方便，例如Like方法用于自定义通配表达式，Equals用于相比较是否相等。

Like

自定义的通配表达式。%表示零长度或任意长度的字符串；_表示一个字符；[]表示在某范围区间的一个字符；[^]表示不在某范围区间的一个字符。比如查询消费者ID以“C”开头的消费者。

```
var q = from c in db.Customers
        where SqlMethods.Like(c.CustomerID, "C%")
        select c;
```

比如查询消费者ID没有“AXOXT”形式的消费者：

```
var q = from c in db.Customers
        where !SqlMethods.Like(c.CustomerID, "A_O_T")
        select c;
```

DateDiffDay

说明：在两个变量之间比较。分别有：DateDiffDay、DateDiffHour、DateDiffMillisecond、DateDiffMinute、DateDiffMonth、DateDiffSecond、DateDiffYear

```
var q = from o in db.Orders
        where SqlMethods
        .DateDiffDay(o.OrderDate, o.ShippedDate) < 10
        select o;
```

语句描述：查询在创建订单后的 10 天内已发货的所有订单。

已编译查询操作(Compiled Query)

说明：在之前我们没有好的方法对写出的SQL语句进行编辑重新查询，现在我们可以这样做，看下面一个例子：

```
//1.创建compiled query
NorthwindDataContext db = new NorthwindDataContext();
var fn = CompiledQuery.Compile(
    (NorthwindDataContext db2, string city) =>
    from c in db2.Customers
    where c.City == city
    select c);
//2.查询城市为London的消费者,用LonCusts集合表示，这时可以用数据控件绑定
var LonCusts = fn(db, "London");
//3.查询城市为Seattle的消费者
var SeaCusts = fn(db, "Seattle");
```

语句描述：这个例子创建一个已编译查询，然后使用它检索输入城市的客户。

LINQ to SQL语句(10)之Insert

1. 简单形式

说明：new一个对象，使用InsertOnSubmit方法将其加入到对应的集合中，使用SubmitChanges()提交到数据库。

```
NorthwindDataContext db = new NorthwindDataContext();
var newCustomer = new Customer
{
    CustomerID = "MCSFT",
    CompanyName = "Microsoft",
    ContactName = "John Doe",
    ContactTitle = "Sales Manager",
    Address = "1 Microsoft Way",
    City = "Redmond",
    Region = "WA",
    PostalCode = "98052",
    Country = "USA",
    Phone = "(425) 555-1234",
    Fax = null
};
db.Customers.InsertOnSubmit(newCustomer);
db.SubmitChanges();
```

语句描述：使用InsertOnSubmit方法将新客户添加到Customers 表对象。调用SubmitChanges 将此新Customer保存到数据库。

2. 一对多关系

说明：Category与Product是一对多的关系，提交Category(一端)的数据时，LINQ to SQL会自动将Product(多端)的数据一起提交。

```
var newCategory = new Category
{
    CategoryName = "Widgets",
    Description = "Widgets are the ……"
};
var newProduct = new Product
{
    ProductName = "Blue Widget",
    UnitPrice = 34.56M,
    Category = newCategory
};
db.Categories.InsertOnSubmit(newCategory);
db.SubmitChanges();
```

语句描述：使用InsertOnSubmit方法将新类别添加到Categories表中，并将新Product对象添加到与此新Category有外键关系的Products表中。调用SubmitChanges将这些新对象及其关系保存到数据库。

3. 多对多关系

说明：在多对多关系中，我们需要依次提交。

```
var newEmployee = new Employee
{
    FirstName = "Kira",
    LastName = "Smith"
};
var newTerritory = new Territory
{
    TerritoryID = "12345",
    TerritoryDescription = "Anytown",
    Region = db.Regions.First()
};
var newEmployeeTerritory = new EmployeeTerritory
{
    Employee = newEmployee,
    Territory = newTerritory
};
db.Employees.InsertOnSubmit(newEmployee);
db.Territories.InsertOnSubmit(newTerritory);
db.EmployeeTerritories.InsertOnSubmit(newEmployeeTerritory);
db.SubmitChanges();
```

语句描述：使用InsertOnSubmit方法将新雇员添加到Employees 表中，将新Territory添加到Territories表中，并将新EmployeeTerritory对象添加到与此新Employee对象和新Territory对象有外键关系的EmployeeTerritories表中。调用SubmitChanges将这些新对象及其关系保持到数据库。

4. 使用动态CUD重写(Override using Dynamic CUD)

说明：CUD就是Create、Update、Delete的缩写。下面的例子就是新建一个ID(主键)为32的Region，不考虑数据库中有没有ID为32的数据，如果有则替换原来的数据，没有则插入。

```
Region nwRegion = new Region()
{
    RegionID = 32,
    RegionDescription = "Rainy"
};
db.Regions.InsertOnSubmit(nwRegion);
db.SubmitChanges();
```

语句描述：使用DataContext提供的分部方法InsertRegion插入一个区域。对SubmitChanges 的调用调用InsertRegion 重写，后者使用动态CUD运行Linq To SQL生成的默认SQL查询。

LINQ to SQL语句(11)之Update

说明：更新操作，先获取对象，进行修改操作之后，直接调用SubmitChanges()方法即可提交。注意，这里是在同一个DataContext中，对于不同的DataContex看下面的讲解。

1. 简单形式

```
Customer cust =
    db.Customers.First(c => c.CustomerID == "ALFKI");
cust.ContactTitle = "Vice President";
db.SubmitChanges();
```

语句描述：使用SubmitChanges将对检索到的一个Customer对象做出的更新保持回数据库。

2. 多项更改

```
var q = from p in db.Products
        where p.CategoryID == 1
        select p;
foreach (var p in q)
{
    p.UnitPrice += 1.00M;
}
db.SubmitChanges();
```

语句描述：使用SubmitChanges将对检索到的进行的更新保持回数据库。

LINQ to SQL语句(12)之Delete和使用Attach

1. 简单形式

说明：调用DeleteOnSubmit方法即可。

```
OrderDetail orderDetail =
    db.OrderDetails.First
    (c => c.OrderID == 10255 && c.ProductID == 36);
db.OrderDetails.DeleteOnSubmit(orderDetail);
db.SubmitChanges();
```

语句描述：使用DeleteOnSubmit方法从OrderDetail 表中删除OrderDetail对象。调用SubmitChanges 将此删除保持到数据库。

2. 一对多关系

说明：Order与OrderDetail是一对多关系，首先DeleteOnSubmit其OrderDetail(多端)，其次DeleteOnSubmit其Order(一端)。因为一端是主键。

```
var orderDetails =
    from o in db.OrderDetails
    where o.Order.CustomerID == "WARTH" &&
    o.Order.EmployeeID == 3
    select o;
var order =
    (from o in db.Orders
     where o.CustomerID == "WARTH" && o.EmployeeID == 3
     select o).First();
foreach (OrderDetail od in orderDetails)
{
    db.OrderDetails.DeleteOnSubmit(od);
}
db.Orders.DeleteOnSubmit(order);
db.SubmitChanges();
```

语句描述语句描述：使用DeleteOnSubmit方法从Order 和Order Details表中删除Order和Order Detail对象。首先从Order Details删除，然后从Orders删除。调用SubmitChanges将此删除保持到数据库。

3. 推理删除(Inferred Delete)

说明：Order与OrderDetail是一对多关系，在上面的例子，我们全部删除CustomerID为WARTH和EmployeeID为3 的数据，那么我们不须全部删除呢？例如Order的OrderID为10248的OrderDetail有很多，但是我们只要删除ProductID为11的OrderDetail。这时就用Remove方法。

```
Order order = db.Orders.First(x => x.OrderID == 10248);
OrderDetail od =
    order.OrderDetails.First(d => d.ProductID == 11);
order.OrderDetails.Remove(od);
db.SubmitChanges();
```

语句描述语句描述：这个例子说明在实体对象的引用实体将该对象从其EntitySet 中移除时，推理删除如何导致在该对象上发生实际的删除操作。仅当实体的关联映射将DeleteOnNull设置为true且CanBeNull 为false 时，才会发生推理删除行为。

使用Attach更新(Update with Attach)

说明：在对于在不同的DataContext之间，使用Attach方法来更新数据。例如在一个名为tempdb的NorthwindDataContext中，查询出Customer和Order，在另一个NorthwindDataContext中，Customer的地址更新为123 First Ave，Order的CustomerID 更新为CHOPS。

```
//通常，通过从其他层反序列化 XML 来获取要附加的实体
//不支持将实体从一个DataContext附加到另一个DataContext
//因此若要复制反序列化实体的操作，将在此处重新创建这些实体
Customer c1;
List<Order> deserializedOrders = new List<Order>();
Customer deserializedC1;
using (NorthwindDataContext tempdb = new NorthwindDataContext())
{
    c1 = tempdb.Customers.Single(c => c.CustomerID == "ALFKI");
    deserializedC1 = new Customer
    {
        Address = c1.Address,
        City = c1.City,
        CompanyName = c1.CompanyName,
        ContactName = c1.ContactName,
        ContactTitle = c1.ContactTitle,
        Country = c1.Country,
        CustomerID = c1.CustomerID,
        Fax = c1.Fax,
        Phone = c1.Phone,
        PostalCode = c1.PostalCode,
        Region = c1.Region
    };
    Customer tempcust =
        tempdb.Customers.Single(c => c.CustomerID == "ANTON");
    foreach (Order o in tempcust.Orders)
```

```
    {
        deserializedOrders.Add(new Order
        {
            CustomerID = o.CustomerID,
            EmployeeID = o.EmployeeID,
            Freight = o.Freight,
            OrderDate = o.OrderDate,
            OrderID = o.OrderID,
            RequiredDate = o.RequiredDate,
            ShipAddress = o.ShipAddress,
            ShipCity = o.ShipCity,
            ShipName = o.ShipName,
            ShipCountry = o.ShipCountry,
            ShippedDate = o.ShippedDate,
            ShipPostalCode = o.ShipPostalCode,
            ShipRegion = o.ShipRegion,
            ShipVia = o.ShipVia
        });
    }
}
using (NorthwindDataContext db2 = new NorthwindDataContext())
{
    //将第一个实体附加到当前数据上下文，以跟踪更改
    //对Customer更新，不能写错
    db2.Customers.Attach(deserializedC1);
    //更改所跟踪的实体
    deserializedC1.Address = "123 First Ave";
    //附加订单列表中的所有实体
    db2.Orders.AttachAll(deserializedOrders);
    //将订单更新为属于其他客户
    foreach (Order o in deserializedOrders)
    {
        o.CustomerID = "CHOPS";
    }
    //在当前数据上下文中提交更改
    db2.SubmitChanges();
}
```

语句描述：从另一个层中获取实体，使用Attach和AttachAll将反序列化后的实体附加到数据上下文，然后更新实体。更改被提交到数据库。

使用Attach更新和删除(Update and Delete with Attach)

说明：在不同的DataContext中，实现插入、更新、删除。看下面的一个例子：

```
//通常，通过从其他层反序列化XML获取要附加的实体
//此示例使用 LoadWith 在一个查询中预先加载客户和订单，
//并禁用延迟加载
Customer cust = null;
using (NorthwindDataContext tempdb = new NorthwindDataContext())
{
    DataLoadOptions shape = new DataLoadOptions();
    shape.LoadWith<Customer>(c => c.Orders);
    //加载第一个客户实体及其订单
    tempdb.LoadOptions = shape;
    tempdb.DeferredLoadingEnabled = false;
    cust = tempdb.Customers.First(x => x.CustomerID == "ALFKI");
}
Order orderA = cust.Orders.First();
Order orderB = cust.Orders.First(x => x.OrderID > orderA.OrderID);
using (NorthwindDataContext db2 = new NorthwindDataContext())
{
    //将第一个实体附加到当前数据上下文，以跟踪更改
    db2.Customers.Attach(cust);
    //附加相关订单以进行跟踪；否则将在提交时插入它们
    db2.Orders.AttachAll(cust.Orders.ToList());
    //更新客户的Phone.
    cust.Phone = "2345 5436";
    //更新第一个订单OrderA的ShipCity.
    orderA.ShipCity = "Redmond";
    //移除第二个订单OrderB.
    cust.Orders.Remove(orderB);
    //添加一个新的订单Order到客户Customer中.
    Order orderC = new Order() { ShipCity = "New York" };
    cust.Orders.Add(orderC);
    //提交执行
    db2.SubmitChanges();
}
```

语句描述：从一个上下文提取实体，并使用 Attach 和 AttachAll 附加来自其他上下文的实体，然后更新这两个实体，删除一个实体，添加另一个实体。更改被提交到数据库。

LINQ to SQL语句(13)之开放式并发控制和事务

Simultaneous Changes开放式并发控制

下表介绍 LINQ to SQL 文档中涉及开放式并发的术语：

术语	说明

并发	两个或更多用户同时尝试更新同一数据库行的情形。
并发冲突	两个或更多用户同时尝试向一行的一列或多列提交冲突值的情形。
并发控制	用于解决并发冲突的技术。
开放式并发控制	先调查其他事务是否已更改了行中的值，再允许提交更改的技术。相比之下，保守式并发控制则是通过锁定记录来避免发生并发冲突。之所以称作开放式控制，是因为它将一个事务干扰另一事务视为不太可能发生。
冲突解决	通过重新查询数据库刷新出现冲突的项，然后协调差异的过程。刷新对象时，LINQ to SQL 更改跟踪器会保留以下数据： 最初从数据库获取并用于更新检查的值 通过后续查询获得的新数据库值。 LINQ to SQL 随后会确定相应对象是否发生冲突（即它的一个或多个成员值是否已发生更改）。如果此对象发生冲突，LINQ to SQL 下一步会确定它的哪些成员发生冲突。LINQ to SQL 发现的任何成员冲突都会添加到冲突列表中。

在 LINQ to SQL 对象模型中，当以下两个条件都得到满足时，就会发生“开放式并发冲突”：客户端尝试向数据库提交更改；数据库中的一个或多个更新检查值自客户端上次读取它们以来已得到更新。 此冲突的解决过程包括查明对象的哪些成员发生冲突，然后决定您希望如何进行处理。

开放式并发(Optimistic Concurrency)

说明：这个例子中在你读取数据之前，另外一个用户已经修改并提交更新了这个数据，所以不会出现冲突。

```
//我们打开一个新的连接来模拟另外一个用户
NorthwindDataContext otherUser_db = new NorthwindDataContext();
var otherUser_product =
    otherUser_db.Products.First(p => p.ProductID == 1);
otherUser_product.UnitPrice = 999.99M;
otherUser_db.SubmitChanges();
//我们当前连接
var product = db.Products.First(p => p.ProductID == 1);
product.UnitPrice = 777.77M;
try
{
    db.SubmitChanges();//当前连接执行成功
}
catch (ChangeConflictException)
{
}
```

说明：我们读取数据之后，另外一个用户获取并提交更新了这个数据，这时，我们更新这个数据时，引起了一个并发冲突。系统发生回滚，允许你可以从数据库检索新更新的数据，并决定如何继续进行您自己的更新。

```
//当前用户
var product = db.Products.First(p => p.ProductID == 1);
//我们打开一个新的连接来模拟另外一个用户
NorthwindDataContext otherUser_db = new NorthwindDataContext() ;
var otherUser_product =
    otherUser_db.Products.First(p => p.ProductID == 1);
otherUser_product.UnitPrice = 999.99M;
otherUser_db.SubmitChanges();
//当前用户修改
product.UnitPrice = 777.77M;
try
{
    db.SubmitChanges();
}
catch (ChangeConflictException)
{
    //发生异常!
}
```

Transactions事务

LINQ to SQL 支持三种事务模型，分别是：

- **显式本地事务**：调用 SubmitChanges 时，如果 Transaction 属性设置为事务，则在同一事务的上下文中执行 SubmitChanges 调用。成功执行事务后，要由您来提交或回滚事务。与事务对应的连接必须与用于构造 DataContext 的连接匹配。如果使用其他连接，则会引发异常。
- **显式可分发事务**：可以在当前 Transaction 的作用域中调用 LINQ to SQL API（包括但不限于 SubmitChanges）。LINQ to SQL 检测到调用是在事务的作用域内，因而不会创建新的事务。在这种情况下，<token>vbtecdlinq</token> 还会避免关闭连接。您可以在此类事务的上下文中执行查询和 SubmitChanges 操作。
- **隐式事务**：当您调用 SubmitChanges 时，LINQ to SQL 会检查此调用是否在 Transaction 的作用域内或者 Transaction 属性是否设置为由用户启动的本地事务。如果这两个事务它均未找到，则 LINQ to SQL 启动本地事务，并使用此事务执行所生成的 SQL 命令。当所有 SQL 命令均已成功执行完毕时，LINQ to SQL 提交本地事务并返回。

1. Implicit（隐式）

说明：这个例子在执行SubmitChanges()操作时，隐式地使用了事务。因为在更新2种产品的库存数量时，第二个产品库存数量为负数了,违反了服务器上的 CHECK 约束。这导致

了更新产品全部失败了，系统回滚到这个操作的初始状态。

```
try
{
    Product prod1 = db.Products.First(p => p.ProductID == 4);
    Product prod2 = db.Products.First(p => p.ProductID == 5);
    prod1.UnitsInStock -= 3;
    prod2.UnitsInStock -= 5;//错误:库存数量的单位不能是负数
    //要么全部成功要么全部失败
    db.SubmitChanges();
}
catch (System.Data.SqlClient.SqlException e)
{
    //执行异常处理
}
```

2.Explicit（显式）

说明：这个例子使用显式事务。通过在事务中加入对数据的读取以防止出现开放式并发异常，显式事务可以提供更多的保护。如同上一个查询中，更新 prod2 的 UnitsInStock 字段将使该字段为负值，而这违反了数据库中的 CHECK 约束。这导致更新这两个产品的事务失败，此时将回滚所有更改。

```
using (TransactionScope ts = new TransactionScope())
{
    try
    {
        Product prod1 = db.Products.First(p => p.ProductID == 4);
        Product prod2 = db.Products.First(p => p.ProductID == 5);
        prod1.UnitsInStock -= 3;
        prod2.UnitsInStock -= 5;//错误:库存数量的单位不能是负数
        db.SubmitChanges();
    }
    catch (System.Data.SqlClient.SqlException e)
    {
        //执行异常处理
    }
}
```

LINQ to SQL语句(14)之Null语义和DateTime

Null语义

说明：下面第一个例子说明查询ReportsToEmployee为null的雇员。第二个例子使用Nullable<T>.HasValue查询雇员，其结果与第一个例子相同。在第三个例子中，使用Nullable<T>.Value来返回ReportsToEmployee不为null的雇员的ReportsTo的值。

1. Null

查找不隶属于另一个雇员的所有雇员：

```
var q =
    from e in db.Employees
    where e.ReportsToEmployee == null
    select e;
```

2.Nullable<T>.HasValue

查找不隶属于另一个雇员的所有雇员：

```
var q =
    from e in db.Employees
    where !e.ReportsTo.HasValue
    select e;
```

3.Nullable<T>.Value

返回前者的EmployeeID 编号。请注意.Value 为可选：

```
var q =
    from e in db.Employees
    where e.ReportsTo.HasValue
    select new
    {
        e.FirstName,
        e.LastName,
        ReportsTo = e.ReportsTo.Value
    };
```

日期函数

LINQ to SQL支持以下DateTime方法。但是，SQL Server和CLR的DateTime类型在范围和计时周期精度上不同，如下表。

类型	最小值	最大值	计时周期
System.DateTime	0001 年 1 月 1 日	9999 年 12 月 31 日	100 毫微秒 （0.0000001 秒）
T-SQL DateTime	1753 年 1 月 1 日	9999 年 12 月 31 日	3.33... 毫秒 （0.0033333 秒）
T-SQL SmallDateTime	1900 年 1 月 1 日	2079 年 6 月 6 日	1 分钟 （60 秒）

CLR DateTime 类型与SQL Server类型相比，前者范围更大、精度更高。因此来自SQL Server的数据用CLR类型表示时，绝不会损失量值或精度。但如果反过来的话，则范围可

能会减小，精度可能会降低；SQL Server日期不存在TimeZone概念，而在CLR中支持这个功能。 我们在LINQ to SQL查询使用以当地时间、UTC 或固定时间要自己执行转换。下面用三个实例说明一下。

1. DateTime.Year

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Year == 1997
    select o;
```

语句描述：这个例子使用DateTime 的Year 属性查找1997 年下的订单。

2. DateTime.Month

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Month == 12
    select o;
```

语句描述：这个例子使用DateTime的Month属性查找十二月下的订单。

3. DateTime.Day

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Day == 31
    select o;
```

语句描述：这个例子使用DateTime的Day属性查找某月 31 日下的订单。

LINQ to SQL语句(15)之String

LINQ to SQL支持以下String方法。但是不同的是默认情况下System.String方法区分大小写。而SQL则不区分大小写。

1. 字符串串联(String Concatenation)

```
var q =
    from c in db.Customers
    select new
    {
        c.CustomerID,
        Location = c.City + ", " + c.Country
    };
```

语句描述：这个例子使用+运算符在形成经计算得出的客户Location值过程中将字符串字段和字符串串联在一起。

2. String.Length

```
var q =
    from p in db.Products
```

```
    where p.ProductName.Length < 10
    select p;
```

语句描述：这个例子使用Length属性查找名称短于10个字符的所有产品。

3. String.Contains(substring)

```
var q =
    from c in db.Customers
    where c.ContactName.Contains("Anders")
    select c;
```

语句描述：这个例子使用Contains方法查找所有其联系人姓名中包含“Anders”的客户。

4. String.IndexOf(substring)

```
var q =
    from c in db.Customers
    select new
    {
        c.ContactName,
        SpacePos = c.ContactName.IndexOf(" ")
    };
```

语句描述：这个例子使用IndexOf方法查找每个客户联系人姓名中出现第一个空格的位置。

5. String.StartsWith(prefix)

```
var q =
    from c in db.Customers
    where c.ContactName.StartsWith("Maria")
    select c;
```

语句描述：这个例子使用StartsWith方法查找联系人姓名以“Maria”开头的客户。

6. String.EndsWith(suffix)

```
var q =
    from c in db.Customers
    where c.ContactName.EndsWith("Anders")
    select c;
```

语句描述：这个例子使用EndsWith方法查找联系人姓名以“Anders”结尾的客户。

7. String.Substring(start)

```
var q =
    from p in db.Products
    select p.ProductName.Substring(3);
```

语句描述：这个例子使用Substring方法返回产品名称中从第四个字母开始的部分。

8. String.Substring(start, length)

```
var q =
    from e in db.Employees
```

```
    where e.HomePhone.Substring(6, 3) == "555"
    select e;
```

语句描述：这个例子使用Substring方法查找家庭电话号码第七位到第九位是“555”的雇员。

9. String.ToUpper()

```
var q =
    from e in db.Employees
    select new
    {
        LastName = e.LastName.ToUpper(),
        e.FirstName
    };
```

语句描述：这个例子使用ToUpper方法返回姓氏已转换为大写的雇员姓名。

10. String.ToLower()

```
var q =
    from c in db.Categories
    select c.CategoryName.ToLower();
```

语句描述：这个例子使用ToLower方法返回已转换为小写的类别名称。

11. String.Trim()

```
var q =
    from e in db.Employees
    select e.HomePhone.Substring(0, 5).Trim();
```

语句描述：这个例子使用Trim方法返回雇员家庭电话号码的前五位，并移除前导和尾随空格。

12. String.Insert(pos, str)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ")"
    select e.HomePhone.Insert(5, ":");
```

语句描述：这个例子使用Insert方法返回第五位为）的雇员电话号码的序列，并在）后面插入一个：。

13. String.Remove(start)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ")"
    select e.HomePhone.Remove(9);
```

语句描述：这个例子使用Remove方法返回第五位为）的雇员电话号码的序列，并移除从第十个字符开始的所有字符。

14. String.Remove(start, length)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ")"
    select e.HomePhone.Remove(0, 6);
```

语句描述：这个例子使用Remove方法返回第五位为 ）的雇员电话号码的序列，并移除前六个字符。

15. String.Replace(find, replace)

```
var q =
    from s in db.Suppliers
    select new
    {
        s.CompanyName,
        Country = s.Country
        .Replace("UK", "United Kingdom")
        .Replace("USA", "United States of America")
    };
```

语句描述：这个例子使用 Replace 方法返回 Country 字段中UK 被替换为 United Kingdom 以及USA 被替换为 United States of America 的供应商信息。

LINQ to SQL语句(16)之对象标识

对象标识

- 运行库中的对象具有唯一标识。引用同一对象的两个变量实际上是引用此对象的同一实例。你更改一个变量后，可以通过另一个变量看到这些更改。
- 关系数据库表中的行不具有唯一标识。由于每一行都具有唯一的主键，因此任何两行都不会共用同一键值。

实际上，通常我们是将数据从数据库中提取出来放入另一层中，应用程序在该层对数据进行处理。这就是 LINQ to SQL 支持的模型。将数据作为行从数据库中提取出来时，你不期望表示相同数据的两行实际上对应于相同的行实例。如果您查询特定客户两次，您将获得两行数据。每一行包含相同的信息。

对于对象。你期望在你反复向 DataContext 索取相同的信息时，它实际上会为你提供同一对象实例。你将它们设计为层次结构或关系图。你希望像检索实物一样检索它们，而不希望仅仅因为你多次索要同一内容而收到大量的复制实例。

在 LINQ to SQL 中，DataContext 管理对象标识。只要你从数据库中检索新行，该行就会由其主键记录到标识表中，并且会创建一个新的对象。只要您检索该行，就会将原始对象实例传递回应用程序。通过这种方式，DataContext 将数据库看到的标识（即主键）

的概念转换成相应语言看到的标识（即实例）的概念。应用程序只看到处于第一次检索时的状态的对象。新数据如果不同，则会被丢弃。

LINQ to SQL 使用此方法来管理本地对象的完整性，以支持开放式更新。由于在最初创建对象后唯一发生的更改是由应用程序做出的，因此应用程序的意向是很明确的。如果在中间阶段外部某一方做了更改，则在调用 SubmitChanges() 时会识别出这些更改。

以上来自MSDN，的确，看了有点"正规"，下面我用两个例子说明一下。

对象缓存

在第一个示例中，如果我们执行同一查询两次，则每次都会收到对内存中同一对象的引用。很明显，cust1和cust2是同一个对象引用。

```
Customer cust1 = db.Customers.First(c => c.CustomerID == "BONAP");
Customer cust2 = db.Customers.First(c => c.CustomerID == "BONAP");
```

下面的示例中，如果您执行返回数据库中同一行的不同查询，则您每次都会收到对内存中同一对象的引用。cust1和cust2是同一个对象引用，但是数据库查询了两次。

```
Customer cust1 = db.Customers.First(c => c.CustomerID == "BONAP");
Customer cust2 = (
    from o in db.Orders
    where o.Customer.CustomerID == "BONAP"
    select o )
    .First()
    .Customer;
```

LINQ to SQL语句(17)之对象加载

对象加载

延迟加载

在查询某对象时，实际上你只查询该对象。不会同时自动获取这个对象。这就是延迟加载。

例如，您可能需要查看客户数据和订单数据。你最初不一定需要检索与每个客户有关的所有订单数据。其优点是你可以使用延迟加载将额外信息的检索操作延迟到你确实需要检索它们时再进行。请看下面的示例：检索出来CustomerID，就根据这个ID查询出OrderID。

```
var custs =
     from c in db.Customers
     where c.City == "Sao Paulo"
     select c;
//上面的查询句法不会导致语句立即执行,仅仅是一个描述性的语句,
只有需要的时候才会执行它
foreach (var cust in custs)
{
```

```
    foreach (var ord in cust.Orders)
    {
        //同时查看客户数据和订单数据
    }
}
```

语句描述：原始查询未请求数据，在所检索到各个对象的链接中导航如何能导致触发对数据库的新查询。

预先加载：LoadWith 方法

你如果想要同时查询出一些对象的集合的方法。LINQ to SQL 提供了DataLoadOptions用于立即加载对象。方法包括： LoadWith 方法，用于立即加载与主目标相关的数据。 AssociateWith 方法，用于筛选为特定关系检索到的对象。

使用 LoadWith方法指定应同时检索与主目标相关的哪些数据。例如，如果你知道你需要有关客户的订单的信息，则可以使用 LoadWith 来确保在检索客户信息的同时检索订单信息。使用此方法可仅访问一次数据库，但同时获取两组信息。 在下面的示例中，我们通过设置DataLoadOptions，来指示DataContext在加载Customers的同时把对应的Orders一起加载，在执行查询时会检索位于Sao Paulo的所有 Customers 的所有 Orders。这样一来，连续访问 Customer 对象的 Orders 属性不会触发新的数据库查询。在执行时生成的SQL语句使用了左连接。

```
NorthwindDataContext db = new NorthwindDataContext();
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(p => p.Orders);
db.LoadOptions = ds;
var custs = (
     from c in db2.Customers
     where c.City == "Sao Paulo"
     select c);
foreach (var cust in custs)
{
    foreach (var ord in cust.Orders)
    {
        Console.WriteLine("CustomerID {0} has an OrderID {1}.",
            cust.CustomerID,
            ord.OrderID);
    }
}
```

语句描述：在原始查询过程中使用 LoadWith 请求相关数据，以便稍后在检索到的各个对象中导航时不需要对数据库进行额外的往返。

延迟加载：AssociateWith方法

使用 AssociateWith 方法指定子查询以限制检索的数据量。 在下面的示例中，AssociateWith 方法将检索的 Orders 限制为当天尚未装运的那些 Orders。如果没有此方法，则会检索所有 Orders，即使只需要一个子集。但是生成SQL语句会发现生成了很多SQL语句。

```
NorthwindDataContext db2 = new NorthwindDataContext();
DataLoadOptions ds = new DataLoadOptions();
ds.AssociateWith<Customer>(
     p => p.Orders.Where(o => o.ShipVia > 1));
db2.LoadOptions = ds;
var custs =
     from c in db2.Customers
     where c.City == "London"
     select c;
foreach (var cust in custs)
{
    foreach (var ord in cust.Orders)
    {
        foreach (var orderDetail in ord.OrderDetails)
        {
           //可以查询出cust.CustomerID, ord.OrderID, ord.ShipVia,
           //orderDetail.ProductID, orderDetail.Product.ProductName
        }
    }
}
```

语句描述：原始查询未请求数据，在所检索到各个对象的链接中导航如何以触发对数据库的新查询而告终。此示例还说明在延迟加载关系对象时可以使用 Assoicate With 筛选它们。

预先加载：LoadWith方法和Associate With方法

这个例子说明：使用LoadWith方法来确保在检索客户信息的同时检索订单信息，在检索订单信息的同时检索订单详细信息， 仅仅访问一次数据库。即可以在一个查询中检索许多对象。使用Associate With方法来限制订单详细信息的排序规则。

```
NorthwindDataContext db2 = new NorthwindDataContext();
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(p => p.Orders);
ds.LoadWith<Order>(p => p.OrderDetails);
ds.AssociateWith<Order>(
     p => p.OrderDetails.OrderBy(o => o.Quantity));
db2.LoadOptions = ds;
var custs = (
```

```
     from c in db2.Customers
     where c.City == "London"
     select c);
foreach (var cust in custs)
{
    foreach (var ord in cust.Orders)
    {
        foreach (var orderDetail in ord.OrderDetails)
        {
           //查询cust.CustomerID, ord.OrderID
           //orderDetail.ProductID, orderDetail.Quantity
        }
    }
}
```

语句描述：在原始查询过程中使用 LoadWith 请求相关数据，以便稍后在检索到的各个对象中导航时此示例还说明在急切加载关系对象时可以使用 Assoicate With 对它们进行排序。

加载重写

这个例子在Category类里提供了一个LoadProducts分部方法。当产品的类别被加载的时候，就直接优先调用了LoadProducts方法来查询没有货源的产品。

```
private IEnumerable<Product> LoadProducts(Category category)
{
    //在执行LINQ to SQL的时候，这个LoadProducts分部方法
    //优先加载执行，这里用存储过程也可以.
    return this.Products
        .Where(p => p.CategoryID == category.CategoryID)
        .Where(p => !p.Discontinued);
}
```

执行下面的查询时，利用上面方法返回的数据进行下面的操作：

```
NorthwindDataContext db2 = new NorthwindDataContext();
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Category>(p => p.Products);
db2.LoadOptions = ds;
var q = (
     from c in db2.Categories
     where c.CategoryID < 3
     select c);
foreach (var cat in q)
{
    foreach (var prod in cat.Products)
```

```
    {
        //查询cat.CategoryID, prod.ProductID
    }
}
```

语句描述：重写 Category 类中的分部方法 LoadProducts。加载某种类别的产品时，调用 LoadProducts 以加载此类别中未停产的产品。

LINQ to SQL语句(18)之运算符转换

运算符转换

1. AsEnumerable：将类型转换为泛型 IEnumerable

使用 AsEnumerable<TSource> 可返回类型化为泛型 IEnumerable 的参数。在此示例中，LINQ to SQL（使用默认泛型 Query）会尝试将查询转换为 SQL 并在服务器上执行。但 where 子句引用用户定义的客户端方法 (isValidProduct)，此方法无法转换为 SQL。 解决方法是指定 where 的客户端泛型 IEnumerable<T> 实现以替换泛型 IQueryable<T>。可通过调用 AsEnumerable<TSource>运算符来执行此操作。

```
var q =
    from p in db.Products.AsEnumerable()
    where isValidProduct(p)
    select p;
```

语句描述：这个例子就是使用AsEnumerable以便使用Where的客户端IEnumerable实现，而不是默认的IQueryable将在服务器上转换为SQL并执行的默认Query<T>实现。这很有必要，因为Where子句引用了用户定义的客户端方法isValidProduct，该方法不能转换为SQL。

2. ToArray：将序列转换为数组

使用 ToArray <TSource>可从序列创建数组。

```
var q =
    from c in db.Customers
    where c.City == "London"
    select c;
Customer[] qArray = q.ToArray();
```

语句描述：这个例子使用 ToArray 将查询直接计算为数组。

3. ToList：将序列转换为泛型列表

使用 ToList<TSource>可从序列创建泛型列表。下面的示例使用 ToList<TSource>直接将查询的计算结果放入泛型 List<T>。

```
var q =
    from e in db.Employees
    where e.HireDate >= new DateTime(1994, 1, 1)
```

```
    select e;
List<Employee> qList = q.ToList();
```

4. ToDictionary：将序列转化为字典

使用Enumerable.ToDictionary<TSource, TKey>方法可以将序列转化为字典。TSource表示source中的元素的类型；TKey表示keySelector返回的键的类型。其返回一个包含键和值的Dictionary<TKey, TValue>。

```
var q =
    from p in db.Products
    where p.UnitsInStock <= p.ReorderLevel && !p.Discontinued
    select p;
Dictionary<int, Product> qDictionary =
    q.ToDictionary(p => p.ProductID);
foreach (int key in qDictionary.Keys)
{
    Console.WriteLine(key);
}
```

语句描述：这个例子使用 ToDictionary 将查询和键表达式直接键表达式直接计算为 Dictionary<K, T>。

LINQ to SQL语句(19)之ADO.NET与LINQ to SQL

它基于由 ADO.NET 提供程序模型提供的服务。因此，我们可以将 LINQ to SQL 代码与现有的 ADO.NET 应用程序混合在一起，将当前 ADO.NET 解决方案迁移到 LINQ to SQL。

1. 连接

在创建 LINQ to SQL DataContext 时，可以提供现有 ADO.NET 连接。对 DataContext 的所有操作（包括查询）都使用所提供的这个连接。如果此连接已经打开，则在您使用完此连接时，LINQ to SQL 会保持它的打开状态不变。我们始终可以访问此连接，另外还可以使用 Connection 属性自行关闭它。

```
//新建一个标准的ADO.NET连接:
SqlConnection nwindConn = new SqlConnection(connString);
nwindConn.Open();
// ... 其它的ADO.NET数据操作代码... //
//利用现有的ADO.NET连接来创建一个DataContext:
Northwind interop_db = new Northwind(nwindConn);
var orders =
     from o in interop_db.Orders
     where o.Freight > 500.00M
     select o;
```

```
//返回Freight>500.00M的订单
nwindConn.Close();
```

语句描述：这个例子使用预先存在的ADO.NET连接创建Northwind对象，本例中的查询返回运费至少为500.00 的所有订单。

2. 事务

当我们已经启动了自己的数据库事务并且我们希望DataContext 包含在内时，我们可以向 DataContext 提供此事务。 通过 .NET Framework 创建事务的首选方法是使用 TransactionScope 对象。通过使用此方法，我们可以创建跨数据库及其他驻留在内存中的资源管理器执行的分布式事务。事务范围几乎不需要资源就可以启动。它们仅在事务范围内存在多个连接时才将自身提升为分布式事务。

```
using (TransactionScope ts = new TransactionScope())
{
    db.SubmitChanges();
    ts.Complete();
}
```

注意：不能将此方法用于所有数据库。例如，SqlClient 连接在针对 SQL Server 2000 服务器使用时无法提升系统事务。它采取的方法是，只要它发现有使用事务范围的情况，它就会自动向完整的分布式事务登记。

下面用一个例子说明一下事务的使用方法。在这里，也说明了重用 ADO.NET 命令和 DataContext 之间的同一连接。

```
var q =
     from p in db.Products
     where p.ProductID == 3
     select p;
//使用LINQ to SQL查询出来
//新建一个标准的ADO.NET连接:
SqlConnection nwindConn = new SqlConnection(connString);
nwindConn.Open();
//利用现有的ADO.NET连接来创建一个DataContext:
Northwind interop_db = new Northwind(nwindConn);
SqlTransaction nwindTxn = nwindConn.BeginTransaction();
try
{
    SqlCommand cmd = new SqlCommand("UPDATE Products SET"
    +"QuantityPerUnit = 'single item' WHERE ProductID = 3");
    cmd.Connection = nwindConn;
    cmd.Transaction = nwindTxn;
    cmd.ExecuteNonQuery();
    interop_db.Transaction = nwindTxn;
```

```
    Product prod1 = interop_db.Products.First(p => p.ProductID == 4);
    Product prod2 = interop_db.Products.First(p => p.ProductID == 5);
    prod1.UnitsInStock -= 3;
    prod2.UnitsInStock -= 5;//这有一个错误，不能为负数
    interop_db.SubmitChanges();
    nwindTxn.Commit();
}
catch (Exception e)
{
    //如果有一个错误，所有的操作回滚
    Console.WriteLine(e.Message);
}
nwindConn.Close();
```

语句描述：这个例子使用预先存在的 ADO.NET 连接创建 Northwind 对象，然后与此对象共享一个 ADO.NET 事务。此事务既用于通过 ADO.NET 连接执行 SQL 命令，又用于通过 Northwind 对象提交更改。当事务因违反 CHECK 约束而中止时，将回滚所有更改，包括通过 SqlCommand 做出的更改，以及通过Northwind 对象做出的更改。

3.直接执行SQL语句

1. 直接执行SQL查询

如果 LINQ to SQL 查询不足以满足专门任务的需要，我们可以使用 ExecuteQuery 方法来执行 SQL 查询，然后将查询的结果直接转换成对象。

```
var products = db.ExecuteQuery<Product>(
    "SELECT [Product List].ProductID,"+
    "[Product List].ProductName " +
    "FROM Products AS [Product List] " +
    "WHERE [Product List].Discontinued = 0 " +
    "ORDER BY [Product List].ProductName;"
);
```

语句描述：这个例子使用ExecuteQuery<T>执行任意 SQL 查询，并将所得的行映射为 Product 对象的序列。

2. 直接执行SQL命令

采用DataContext 连接时，可以使用ExecuteCommand来执行不返回对象的 SQL 命令。

```
db.ExecuteCommand
    ("UPDATE Products SET UnitPrice = UnitPrice + 1.00");
```

语句描述：使用ExecuteCommand执行任意SQL命令，本例中为将所有产品单价提高 1.00 的批量更新。

LINQ to SQL语句(20)之存储过程

在我们编写程序中，往往需要一些存储过程，在LINQ to SQL中怎么使用呢？也许比原来的更简单些。下面我们以NORTHWND.MDF数据库中自带的几个存储过程来理解一下。

1. 标量返回

在数据库中，有名为Customers Count By Region的存储过程。该存储过程返回顾客所在"WA"区域的数量。

```
ALTER PROCEDURE [dbo].[NonRowset]
    (@param1 NVARCHAR(15))
AS
BEGIN
    SET NOCOUNT ON;
     DECLARE @count int
     SELECT @count = COUNT(*)FROM Customers
     WHERECustomers.Region = @Param1
     RETURN @count
END
```

我们只要把这个存储过程拖到O/R设计器内，它自动生成了以下代码段：

```
[Function(Name = "dbo.[Customers Count By Region]")]
public int Customers_Count_By_Region([Parameter
(DbType = "NVarChar(15)")] string param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
    return ((int)(result.ReturnValue));
}
```

我们需要时，直接调用就可以了，例如：

```
int count = db.CustomersCountByRegion("WA");
Console.WriteLine(count);
```

语句描述：这个实例使用存储过程返回在“WA”地区的客户数。

2. 单一结果集

从数据库中返回行集合，并包含用于筛选结果的输入参数。 当我们执行返回行集合的存储过程时，会用到结果类，它存储从存储过程中返回的结果。

下面的示例表示一个存储过程，该存储过程返回客户行并使用输入参数来仅返回将“London”列为客户城市的那些行的固定几列。

```
ALTER PROCEDURE [dbo].[Customers By City]
     -- Add the parameters for the stored procedure here
     (@param1 NVARCHAR(20))
AS
BEGIN
     -- SET NOCOUNT ON added to prevent extra result sets from
```

```
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    SELECT CustomerID, ContactName, CompanyName, City from
    Customers as c where c.City=@param1
END
```

拖到O/R设计器内，它自动生成了以下代码段：

```
[Function(Name="dbo.[Customers By City]")]
public ISingleResult<Customers_By_CityResult> Customers_By_City(
[Parameter(DbType="NVarChar(20)")] string param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this, (
    (MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
    return ((ISingleResult<Customers_By_CityResult>)
    (result.ReturnValue));
}
```

我们用下面的代码调用：

```
ISingleResult<Customers_By_CityResult> result =
 db.Customers_By_City("London");
foreach (Customers_By_CityResult cust in result)
{
    Console.WriteLine("CustID={0}; City={1}", cust.CustomerID,
        cust.City);
}
```

语句描述：这个实例使用存储过程返回在伦敦的客户的 CustomerID和City。

3. 多个可能形状的单一结果集

当存储过程可以返回多个结果形状时，返回类型无法强类型化为单个投影形状。尽管 LINQ to SQL 可以生成所有可能的投影类型，但它无法获知将以何种顺序返回它们。ResultTypeAttribute 属性适用于返回多个结果类型的存储过程，用以指定该过程可以返回的类型的集合。

在下面的 SQL 代码示例中，结果形状取决于输入（param1 = 1或param1 = 2）。我们不知道先返回哪个投影。

```
ALTER PROCEDURE [dbo].[SingleRowset_MultiShape]
    -- Add the parameters for the stored procedure here
    (@param1 int )
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
```

```
    if(@param1 = 1)
    SELECT * from Customers as c where c.Region = 'WA'
    else if (@param1 = 2)
    SELECT CustomerID, ContactName, CompanyName from
    Customers as c where c.Region = 'WA'
END
```

拖到O/R设计器内，它自动生成了以下代码段：

```
[Function(Name="dbo.[Whole Or Partial Customers Set]")]
public ISingleResult<Whole_Or_Partial_Customers_SetResult>
Whole_Or_Partial_Customers_Set([Parameter(DbType="Int")]
System.Nullable<int> param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
    return ((ISingleResult<Whole_Or_Partial_Customers_SetResult>)
    (result.ReturnValue));
}
```

但是，VS2008会把多结果集存储过程识别为单结果集的存储过程，默认生成的代码我们要手动修改一下，要求返回多个结果集，像这样：

```
[Function(Name="dbo.[Whole Or Partial Customers Set]")]
[ResultType(typeof(WholeCustomersSetResult))]
[ResultType(typeof(PartialCustomersSetResult))]
public IMultipleResults Whole_Or_Partial_Customers_Set([Parameter
(DbType="Int")] System.Nullable<int> param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
    return ((IMultipleResults)(result.ReturnValue));
}
```

我们分别定义了两个分部类，用于指定返回的类型。WholeCustomersSetResult类 如下：

PartialCustomersSetResult类 如下：

这样就可以使用了，下面代码直接调用，分别返回各自的结果集合。

```
//返回全部Customer结果集
IMultipleResults result = db.Whole_Or_Partial_Customers_Set(1);
IEnumerable<WholeCustomersSetResult> shape1 =
 result.GetResult<WholeCustomersSetResult>();
foreach (WholeCustomersSetResult compName in shape1)
{
    Console.WriteLine(compName.CompanyName);
```

```
}
//返回部分Customer结果集
result = db.Whole_Or_Partial_Customers_Set(2);
IEnumerable<PartialCustomersSetResult> shape2 =
 result.GetResult<PartialCustomersSetResult>();
foreach (PartialCustomersSetResult con in shape2)
{
    Console.WriteLine(con.ContactName);
}
```

语句描述：这个实例使用存储过程返回“WA”地区中的一组客户。返回的结果集形状取决于传入的参数。如果参数等于 1，则返回所有客户属性。如果参数等于2，则返回ContactName属性。

4. 多个结果集

这种存储过程可以生成多个结果形状，但我们已经知道结果的返回顺序。

下面是一个按顺序返回多个结果集的存储过程Get Customer And Orders。 返回顾客ID为"SEVES"的顾客和他们所有的订单。

```
ALTER PROCEDURE [dbo].[Get Customer And Orders]
(@CustomerID nchar(5))
    -- Add the parameters for the stored procedure here
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    SELECT * FROM Customers AS c WHERE c.CustomerID = @CustomerID
    SELECT * FROM Orders AS o WHERE o.CustomerID = @CustomerID
END
```

拖到设计器代码如下：

```
[Function(Name="dbo.[Get Customer And Orders]")]
public ISingleResult<Get_Customer_And_OrdersResult>
Get_Customer_And_Orders([Parameter(Name="CustomerID",
DbType="NChar(5)")] string customerID)
{
     IExecuteResult result = this.ExecuteMethodCall(this,
     ((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID);
     return ((ISingleResult<Get_Customer_And_OrdersResult>)
     (result.ReturnValue));
}
```

同样，我们要修改自动生成的代码：

```
[Function(Name="dbo.[Get Customer And Orders]")]
[ResultType(typeof(CustomerResultSet))]
[ResultType(typeof(OrdersResultSet))]
public IMultipleResults Get_Customer_And_Orders
([Parameter(Name="CustomerID",DbType="NChar(5)")]
string customerID)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID);
    return ((IMultipleResults)(result.ReturnValue));
}
```

同样，自己手写类，让其存储过程返回各自的结果集。

CustomerResultSet类

OrdersResultSet类

这时，只要调用就可以了。

```
IMultipleResults result = db.Get_Customer_And_Orders("SEVES");
//返回Customer结果集
IEnumerable<CustomerResultSet> customer =
result.GetResult<CustomerResultSet>();
//返回Orders结果集
IEnumerable<OrdersResultSet> orders =
 result.GetResult<OrdersResultSet>();
//在这里，我们读取CustomerResultSet中的数据
foreach (CustomerResultSet cust in customer)
{
    Console.WriteLine(cust.CustomerID);
}
```

语句描述：这个实例使用存储过程返回客户“SEVES”及其所有订单。

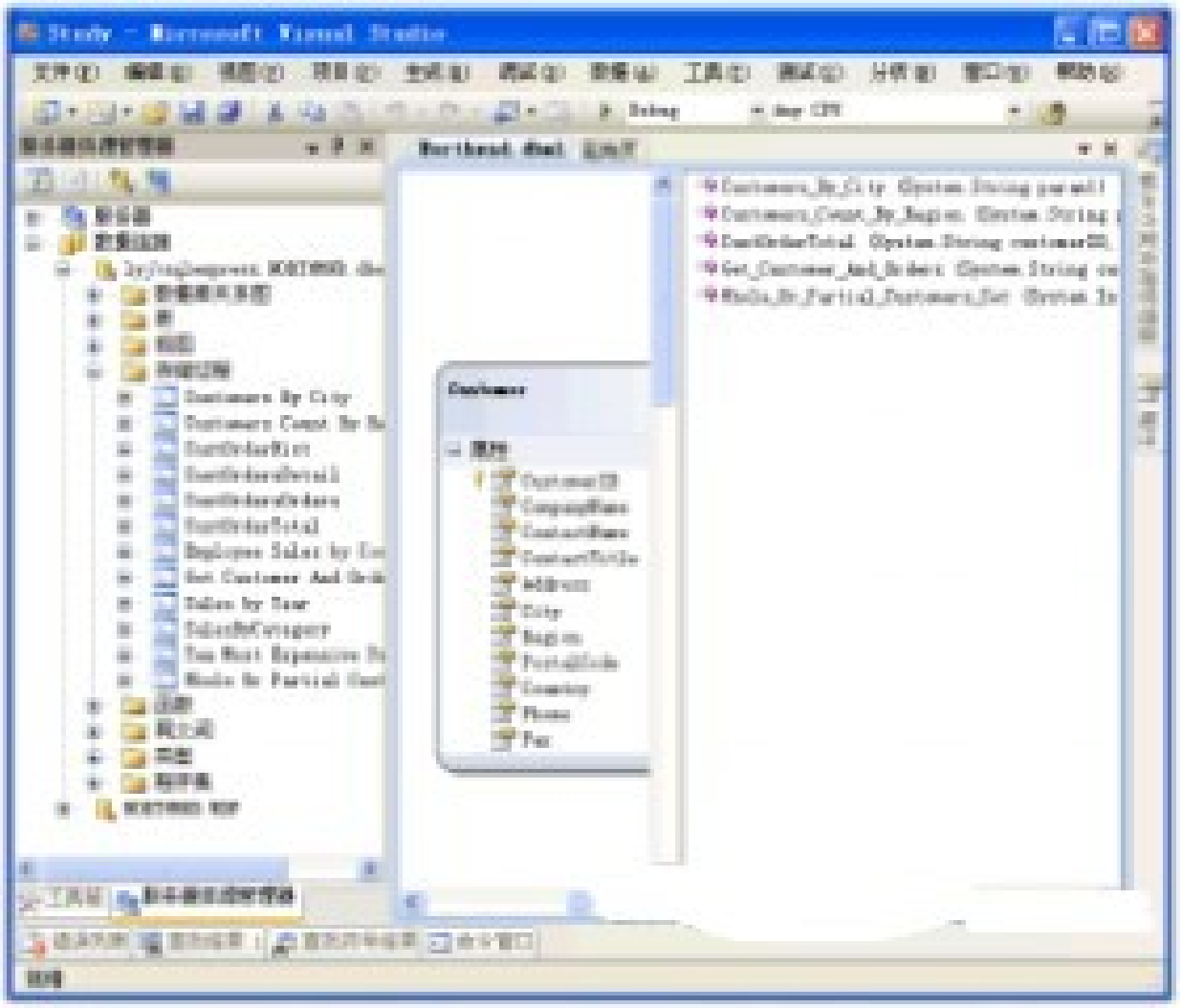
5. 带输出参数

LINQ to SQL 将输出参数映射到引用参数，并且对于值类型，它将参数声明为可以为 null。

下面的示例带有单个输入参数（客户 ID）并返回一个输出参数（该客户的总销售额）。

```
ALTER PROCEDURE [dbo].[CustOrderTotal]
@CustomerID nchar(5),
@TotalSales money OUTPUT
AS
SELECT @TotalSales = SUM(OD.UNITPRICE*(1-OD.DISCOUNT) * OD.QUANTITY)
FROM ORDERS O, "ORDER DETAILS" OD
where O.CUSTOMERID = @CustomerID AND O.ORDERID = OD.ORDERID
```

把这个存储过程拖到设计器中，图片如下：



其生成代码如下：

```
[Function(Name="dbo.CustOrderTotal")]
public int CustOrderTotal(
[Parameter(Name="CustomerID", DbType="NChar(5)")]string customerID,
[Parameter(Name="TotalSales", DbType="Money")]
  ref System.Nullable<decimal> totalSales)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo)(MethodInfo.GetCurrentMethod())),
    customerID, totalSales);
    totalSales = ((System.Nullable<decimal>)
    (result.GetParameterValue(1)));
    return ((int)(result.ReturnValue));
}
```

我们使用下面的语句调用此存储过程：注意：输出参数是按引用传递的，以支持参数为“in/out”的方案。在这种情况下，参数仅为“out”。

```
decimal? totalSales = 0;
string customerID = "ALFKI";
db.CustOrderTotal(customerID, ref totalSales);
Console.WriteLine("Total Sales for Customer '{0}' = {1:C}",
customerID, totalSales);
```

语句描述：这个实例使用返回 Out 参数的存储过程。

好了，就说到这里了，其增删改操作同理。相信大家通过这5个实例理解了存储过程。