I、前言

一. 简介

ALinq Dynamic 为 ALinq 以及 Linq to SQL 提供了一个 Entiy SQL 的查询接口，使得它们能够应用 Entity SQL 进行数据的查询。它的原理是将 Entiy SQL 解释为 Linq 表达式，再执行生成的 Linq 表达式。

1. 关于 Entity SQL

Entity SQL 是一种类似于 SQL 的语言，用于在 Entity Framework 中查询概念模型。概念模型将数据表示为实体和关系，而 Entity SQL 允许您以那些用过 SQL 的人熟悉的格式查询这些实体和关系。
以上这段话，摘自 Entity Framwork 的文档。也就是说，Entity SQL 是由微软的 Entity Framework 开小组设计，并且应用于 Entity Framework。

2. ALinq Dynamic 与 Entity Framework 的关系

ALinq Dynamic 只是将 Entity SQL 移植过来，并遵循 EntitySQL 的语法，使得 Entity SQL 能够应用于 ALinq 和 Linq to SQL 框架，而不仅是 Entity Framwork。

3. Entity SQL 的兼容性

ALinq Dynamic 兼容性绝大部份的 Entity SQL，但由于 Entity SQL 是为 Entity Framework 而设计的，个别针对 Entity Framework 的特定功能并不支持，具体请参阅文档。

二. 软件的授权

ALinq Dynamic 使用的是 MIT 协议授权。

三. 功能特点

1. 兼容 EF 中的 Entity SQL ，并实现了大部份功能。
2. 支持 ALinq 和 Linq to SQL 两种框架。
3. 支持 .NET 3.5 以及更高版本的 .NET 框架。

四. 文档说明

本文档在介绍 ALinq Dynamic 功能时，主要采用与 Entity Framework 作对比来描述。表格说明：
1、表格标题栏中的"A"表示"ALinq Dynamic"，"E"表示"Entity Framework"。
2、"Y"表示支持，"N"表示不支持。
示例：

A	E	函数	说明
Y	Y	**Avg** (exression)	
N	Y	**Var** (expression)	

示例中的表示，ALinq Dynamic 支持 Avg 函数，但是不支持 Var 函数，而 Entity Framework 支持两者。

II、代码的获取与编译

1. 下载代码

ALinq Dynamic 项目托管在 CodePlex 网站，你可以使用浏览器下载压缩包，或者通过 SVN 获取。
项目网址：http://esql.codeplex.com/
压缩包下载网址：http://esql.codeplex.com/releases/
SVN 地址：https://esql.svn.codeplex.com/svn

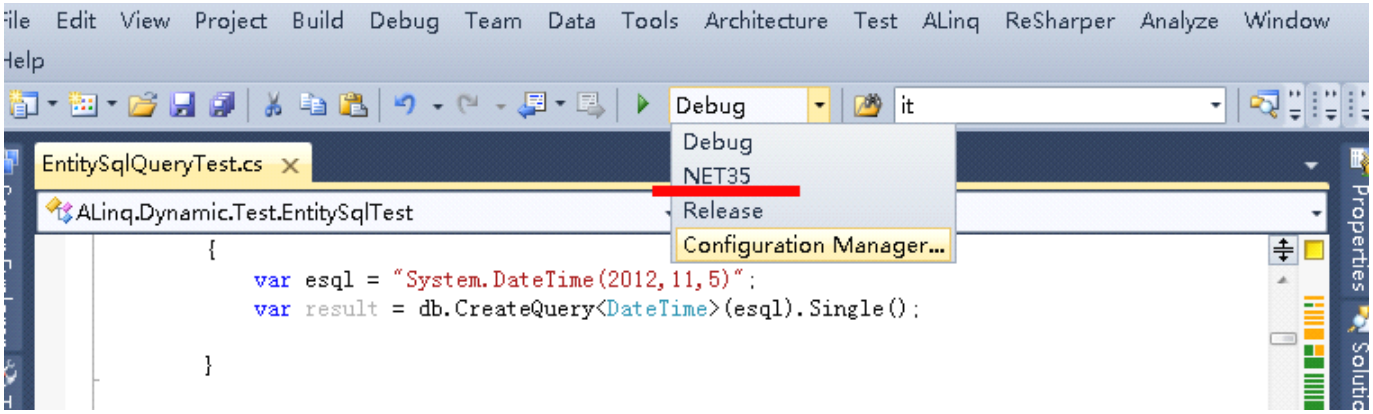
2. 编译与运行代码

代码的编译

对于 ALinq 用户，如果你使用的是 VS 2010 ，请打开 DynamicQuery.sln 项目，如果使用的是 VS2 008，则打开 DynamicQuery_VS08.sln 项目。

对于 Linq to SQL 用户，项目只支持 VS 2010，请打开 DynamicQuery.LinqToSql.sln 项目。

打开项目后，在编译选项里面，有 NET35 的选项，如果你需要支持.NET Framework 35，默认是使用 .NET Framework4。



单元测试的运行

在运行单元测试前，你需要修改数据库的连接字符串，打开 DynamicQueryTest 项目中的 App.Config 文件，对于 ALinq 用户，你需要修改名为 sqliteDB 的数据库连接字符串，对于 Linq to SQL 用户，你需要修改 sqlceDB 的数据库连接字符串。修改完之后，你就可以在单元测试的面板中，运行单运测试示例，如果单元测试的面板没有出现，点击菜单 Test->Windows->TestView

III、概述

一. 使用

1. 程序集与命名空间的引用

使用 ALinq Dynamic，你需要引用 ALinq.Dynamic.dll（ALinq 用户）或者 System.Data.Linq.Dynamic.dll (Linq to SQL 用户），在使用时，还需要引入 ALinq.Dynamic 命名空间。当然，使用前你还需要完成建模的工作，本文假设你已经会了，否则请参考 Linq to SQL 或 ALinq 教程。

示例一

下面的示例，由于使用到 dynamic 关键字，必须运行在.NET Framework4 或以上。

```
using System;
using ALinq.Dynamic;
using NorthwindDemo;
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            var db = new NorthwindDataContext();
            var q = db.CreateQuery("select e.FirstName, e.LastName from Employees as e");
            foreach (var item in q)
            {
                var e = item as dynamic;
                Console.WriteLine("{0} {1}", e.FirstName, e.LastName);
            }
```

```
        }
    }
}
```

示例二

下面的示例可以运行在.NET Framework 3.5（为了便于阅读以及节省编幅，仅给出关键部份代码）。

```
var db = new NorthwindDataContext();
var q = db.CreateQuery<IDataRecord>("select e.FirstName, e.LastName from Employees as e");
foreach (var e in q)
    Console.WriteLine("{0} {1}", e["FirstName"], e["LastName"]);
```

二. 大小敏感

1. 关键词不区分大小写

对关键词是不区分大小写的，例如下面的这两条 ESQL 语句是相等的。

语句一

```
select p from Product as p
```

语句二

```
SELECT p FROM Product AS p
```

2. 命名空间区分大小写

对于命名空间，是区分大小写的，例如下面的两条 ESQL 语句是**不等价**的。

语句一

```
using NorthwindDemo
select p from Product as p
```

语句二

```
using NORTHWINDDEMO
SELECT p FROM Product AS p
```

这是因为命名空间是区分大小区的，NorthwindDemo 和 NORTHWINDDEMO 是不等价的。

3. 实体类区分小写

对于实体类的名称，是区分大小写的，例如下面的例句是**不等价**的

语句一：

```
select p from Product as p
```

语句二：

```
select p from PRODUCT as p
```

这是因为，实体类的名称，Product 和 PRODUCT 不等价。

4. 属性区分大小写

对属性是区分大小写的，例如下面的例句是**不等价**的

语句一：

```
select p.ProductId, p.ProductName from Product as p
```

语句二：

```
select p.ProductID, p.ProductNAME from Product as p
```

5. 函数名称的大小

对于函数名称，有点特殊，如果是系统自带的函数，则不区分大小写，如果是用户自定义函数，则区分大小写。

示例

```
select value GetFullName(e.FirstName, e.LastName) from Employees as e
```

其中 GetFullName 是区分大小写的。

三. 命名空间

1. using 关键字

通过使用 using 关键字，可以引入命名空间，避免使用全局标识。例如：

```
var esql = @"using NorthwindDemo;
             select e.FirstName, e.LastName
             from Employee as e";
var q1 = db.CreateQuery(esql);
```

其等效于：

```
var esql = @"select e.FirstName, e.LastName
             from NorthwindDemo.Employee as e";
var q1 = db.CreateQuery<ALinq.Dynamic.Test.Person2>(esql);
```

2. 默认引用的命名空间

在使用 CreateQuery 方法进行查询的时候，ALinq Dynamic 会自动引入 DataContext 所在的命名空间。例如：

```
var db = new NorthwindDemo.NorthwindDataContext();
var esql = "select e.FirstName, e.LastName from Employee as e";
```

由于 DataContext ，即 db 变量，对应的实体类命名空间为 NorthwindDemo， esql 语句会自动引入 NothwindDemo 命名空间，即等效于下面的语句：

```
var db = new NorthwindDemo.NorthwindDataContext();
var esql = @"using NorthwindDemo;
             select e.FirstName, e.LastName from Employee as e";
```

四. 标识符

1. 简单标识符

标识符由字母，数字，和下划线组成，并且第一个字符必须是字母（a-z 或 A-Z）。

2. 带引号的标识符

带引号的标识符是括在方括号（[]）中的任意字符序列。使用带引号的标识符可以指定含有在标识符中无效的字符的标识符。方括号中的所有字符都是标识符的一部分，包括所有空格。

但是，带引号的标识符，第一个字符不能是数字，并且不能包含以下字符：

换行符
回车符
制表符
Backspace
额外的方括号（即括起标识符的方括号中的方括号）。
单引号（即：'）
双引号（即："）

使用带引号的标识符可以创建在标识符中无效的属性名称字符，如下面的示例所示：

```
var esql = "select c.ContactName as [Contact Name] from Customers as c";
var q = db.CreateQuery(esql).Execute();
```

使用带引号的标识符，还可以指定关键字作为标识符。例如，如果类型 Email 具有名为“From”的属性，则可以使用方括号来消除与保留关键字“FROM”的歧义，如下所示：

```
var esql = "select e.[From] from Emails AS e";
```

但是下面的例子是非法的

标识符中带引号

```
var esql = "select c.ContactName as ['Contact Name'] from Customers as c";
```

标识符以数字开始

```
var esql = "select c.ContactName as [0ContactName] from Customers as c";
```

3. 别名规则

如果需要，建议在查询中指定别名，别名可以应用于：

行构造函数

查询语句的 FROM 子句

查询语句的 SELECT 子句

查询语句的 GROUP BY 子句

有效的别名

有效的别外包括简单标识符或者带引号的标识符

五. 参数

参数是查询语句之外定义的变量，参数名在查询语句中定义，并以(@)符号作为前缀。这可以将它们与属性名或查询中定义的其它名称区分开来。参数为两类，命名参数和顺序参数，对于命名参数，必须指名称，参数的传递不要按照顺序。顺序参数不需要指定名称，在查询语句中使用序号（序号从 0 开始）作为参数名，参数必须按顺序传递。

命名参数

```
var esql = @"select e from Employees as e
             where e.FirstName = @f and e.LastName = @l";
var q = db.CreateQuery(esql, new ObjectParameter("l", "mak"), new ObjectParameter("f", "mike"));
```

顺序参数

```
var esql = @"select e from Employees as e
             where e.FirstName = @0 and e.LastName = @1";
var q = db.CreateQuery(esql, "mike", "mak");
```

六. 变量

变量表达式是对当前作用域中定义的命名表达式的引用。变量引用必须是有效的标识符。

以下示例演示如何在查询语句中使用变量。FROM 子句中的 c 是变量定义。在 SELECT 子句中使用的 c 表示变量引用。

```
var esql = "select c.ContactName from Customers as c";
```

七. 文字

名称	说明
空值	使用 null 表示，空文字用于表示对任何类型均为空的值。
布尔值	布尔值文字由关键字 true 和 false 表示。
整型	整型文字可以为类型 Int32，UInt32，Int64，UInt64。 Int32 文字是一系列数字字符。例如：123 。 UInt32 文字是一系列数字字符，后跟一个 U。例如：123u 。 Int64 文字是一系列数字字符，后跟一个 L。例如：123l 。 UInt64 文字是一系列数字字符，后跟一个 UL。 例如：123ul 。
小数	固定点数字（小数）是一系列数字字符、一个圆点 (.) 和另一系列数字字符，后跟一个字母“M”。例如： 123m
双精度小数	双精度浮点数字是一系列数字字符、一个圆点 (.) 和另一系列数字字符，后面可能跟一个指数。例如:

	123.67，123.67E+3
单精度小数	单精度浮点数字（或浮点数）是双精度浮点数字语法后跟小写 f。 例如：123.67f，123.67F
字符串	字符串是包含在引号内的一系列字符。引号可以同时为单引号 (') 或同时为双引号 (")。例如：'hello'，"This is a string!" 。
日期时间	日期时间文字独立于区域设置并由日期部分和时间部分组成，其中日期部分是必需的。日期部分必须采用格式：YYYY-MM-DD，其中，YYYY 是介于 0001 与 9999 之间的四位年度值，MM 是介于 1 与 12 之间的月份，而 DD 是对给定月份 MM 有效的日期值。 时间部分的格式必须为：HH:MM[:SS]，默认值为 00:00:00，其中 HH 为介于 0 至 23 之间的小时值，MM 为介于 0 至 59 之间的分钟值，SS 为介于 0 至 59 之间的秒值。
二进制	二进制字符串文字是由单引号分隔的一系列十六进制数字，位于关键字“binary”或快捷方式符号 X 或 x 之后。快捷方式符号 X 是不区分大小写的。例如：Binary '00ffaabb'，
Guid	GUID 文本表示全局唯一标识符。该文本是一个序列，由关键字 GUID 后跟十六进制数字构成，采用称为“注册表”格式的格式：包含在单引号内的 8-4-4-4-12。十六进制数字区分大小写。 在 GUID 符号与文字负载之间可以存在任意数目的空格，但是不能存在新行。例如：Guid'CD582A70-863B-4E4B-9530-174A83EEB0C0'，GUID 'CD582A70-863B-4E4B-9530-174A83EEB0C0' 。

八. 运算符优先级

九. 成员访问

使用点运算符（.）可以访问实例的属性、字段、方法。如果成员是元素的集合，还使用方括号（[]）可以访问集合中的元素。

属性访问

```
select o.OrderId, o.OrderDate from Orders as o
select o.OrderDate.Value.Year from Orders as o where o.OrderDate is not null
```

方法访问

将 OrderDate 转为 'yyyy-MM-dd' 格式的字符串。

```
select o.OrderDate.Value.ToString('yyyy-MM-dd') as Date
from Orders as o where o.OrderDate is not null
```

访问集合中的元素

下面的例句中，获取 OrderDetails 属性中的第一个 OrderDetail 。

```
var esql = "select o.OrderId, o.OrderDetails[0] as FirstOrderDetail from Orders as o";
db.CreateQuery(esql).Execute();
```

静态方法的访问

下面的例句中，调用 System.Guid 类的 NewGuid 方法

```
var esql = "select System.Guid.NewGuid() as Guid, e.FirstName from Employees as e";
var q = db.CreateQuery<IDataRecord>(esql);
foreach (var item in q)
    Console.WriteLine("{0} {1}", item[0], item[1]);
```

静态属性的访问

下面的例句中，调用 System.Math 类的 PI 属性

```
var esql = "System.Math.PI";
var pi = db.CreateQuery<float>(esql).Single();
```

DataContext 方法访问

下面的例名中的 GetFullName 是 db 实例的一个方法。

```
var esql = "select value GetFullName(e.FirstName, e.LastName) from Employees as e";
q = db.CreateQuery<string>(esql);
```

DataContext 属性访问

下面的例句演示 DataContext 属性的访问，其中的 Version 是 db 实例的一个属性。

```
var esql = "select Version, e.FirstName from Employees as e";
var q = db.CreateQuery<IDataRecord>(esql);
```

十. 分页

.可以使用 SKIP 和 LIMIT 关键字来执行物理分面，如果只需要限制返回的记录数，也可以使用 TOP 关键字。但 TOP 和 SKIP/LIMIT/TAKE 是互斥的，不能一起使用。

TOP 概述

在 SELECT 关键字之后，可以使用 TOP 关键字，限制返回的记录数。注意：是 TOP (NUMBER) 而不是 TOP NUMBER。正确的例句：

```
var esql = "select top(1) p from Products as p";
db.CreateQuery(esql).Execute();
```

SKIP 和 LIMIT/TAKE 概述

SKIP 和 LIMIT/TAKE 是 SELECT 子句的组成部份，SKIP 和 LIMIT/TAKE 不必一起出现，可以单独使用 SKIP 和 LIMIT/TAKE，但必须出现 SELECT 子语句的最后，另外，LIMIT 与 TAKE是等效的。

示例一:

```
var esql = "select p from Products as p skip 10";
var products = db.CreateQuery<Product>(esql).Execute();
```

示例二：

```
var esql = "select p from Products as p limit 10";
var products = db.CreateQuery<Product>(esql).Execute();
```

或

```
var esql = "select p from Products as p take 10";
var products = db.CreateQuery<Product>(esql).Execute();
```

例句 3：

```
var esql = "select p from Products as p skip 10 limit 10";
var products = db.CreateQuery<Product>(esql).Execute();
```

或

```
var esql = "select p from Products as p skip 10 take 10";
var products = db.CreateQuery<Product>(esql).Execute();
```

十一. 编写查询语句

1. 嵌套的查询语句

嵌套表达式

嵌套表达式可以放置在任何可接受其返回类型值的位置。例如：

```
var esql = @"Row('mike' as FirstName, 'mak' as LastName,
                    Row('ansiboy@163.com' as Email,
                        '13434126607' as Phone) as Contact)";
var row = db.CreateQuery<IDataRecord>(esql).Single() as dynamic;
Console.WriteLine("{0} {1}", row.FirstName, row.LastName);
Console.WriteLine("{0} {1}", row.Contact.Email, row.Contact.Phone);
```

嵌套查询

嵌套查询可以放在投影子句中。例如：

```
var esql = @"select c.CategoryID, c.CategoryName,
                    (select value count() from c.Products) as ProductCount
             from Categories as c
             limit 5";
var items = db.CreateQuery<dynamic>(esql);
foreach (var item in items)
{
    Console.WriteLine("ID:{0}, Name: {1}, Products Count:{2}", item.CategoryID,
                      item.CategoryName, item.ProductCount);
}
```

在查询语句中，嵌套查询必须括在括号中，例如：

```
var esql = @"(select p.ProductID, p.ProductName from Products as p where p.UnitPrice < 100 )
              union
             (select p.ProductID, p.ProductName from Products as p where p.UnitPrice > 200)";
var q = db.CreateQuery<IDataRecord>(esql);
```

2. 使用索引器查询

下面的示例，选取员工的 FirstName 和 LastName，并且过滤掉 FirstName 为 Mike 的记录。

```
var esql = @"select e.FirstName, e.LastName from Employees as e";
var q = db.CreateQuery<IDataRecord>(esql)
            .Where(o=> (string)o["FirstName"] != "Mike");
```

3. 使用接口查询

下面的示例是基于接口的查询示例，值得注意的是，Employee 实体类必须继承 IEmployee 接口，否则会出异常。

```
var esql = "select e from Employees as e";
var q = db.CreateQuery<IEmployee>(esql)
            .Where(o => o.FirstName == "F" && o.LastName == "L")
            .Select(o => new { o.FirstName, o.LastName, o.BirthDate });
```

4. 参数化数据源

使用参数化的数据源，可以将 Linq 查询与 Entity SQL 查询结合起来。下面的示例中，选通过 Linq 查询获取 Country 为 EN 的员工记录，然后在 Entity SQL 语句中，再过滤掉 LastName 为 Mak 的记录。

```
var employees = db.Employees.Where(o => o.Country == "EN");
var esql = "select e from @0 as e where e.LastName != 'Mak'";
var q = db.CreateQuery(esql, employees);
```

IV、参考

一. 算术运算符

A \ E	运算符	说明
Y	Y	+（加）	加法。
Y	Y	/（除）	除法
Y	Y	%（取模）	返回除法运算的余数。
Y	Y	*（乘）	乘法。
Y	Y	-（负号）	求反。
Y	Y	-（减）	减法。

二. 规范函数

1. 聚合函数

A	E	函数	说明
Y	Y	**Avg** (exression)	返回非空的平均值。
Y	Y	**BigCount** ()	返回集合的记录数。
Y	Y	**Count** ()	返回集合的记录数
Y	Y	**Max** (expression)	返回非空值的最大值。
Y	Y	**Min** ([expression])	返回非空值的最小值。
N	Y	**StDev** (expression)	
N	Y	**StDevP** (expression)	
Y	Y	**Sum** (expression)	返回非空值的总和。
N	Y	**Var** (expression)	
N	Y	**VarP** (expression)	

1）Avg

返回非空的平均值。

参数

exression：数值类型

返回值

expression 的类型。

示例

```
select value avg(p.UnitPrice) from Products as p
```

2）BigCount

返回集合的记录数。

示例

```
1. select value bigcount() from products as p
```

3）Count

返回集合的记录数

示例

```
1. select value count() from products as p
2. select value count(c.Products) from category as c
3. select c.CategoryId, count(select p from c.Products as p) as ProductsCount from category as c
```

4）Max

示例

```
1. select value max(p.UnitPrice) from Products as p
2. select max(select value p.UnitPrice from c.Products as p) from Categories as c
```

5）Min

返回非空值的最小值。

示例

```
1. select value min(p.UnitPrice) from Products as p
```

```
2. select min(select value p.UnitPrice from c.Products as p) from Categories as c
```

6）Sum

返回非空值的总和。

示例

```
1. select sum(o.UnitPrice) from OrderDetails as o
2. select key, sum(o.Quantity) as SumQuantity  from OrderDetails as o group by o.Product as key
```

基于集合的聚合

基于集合的聚合（集合函数）针对集合而运行并返回值。 例如，如果 ORDERS 是所有订单的集合，则可以使用以下表达式计算最早的发货日期：

```
select value min(o.ShippedDate) from Orders as o
```

将在当前环境名称解析范围内计算基于集合的聚合内的表达式。

基本组的聚合

基于组的聚合将按照 GROUP BY 子句定义的方式对组进行计算。 对于结果中的每个组，将使用每个组中的元素作为聚合计算的输入来计算单独的聚合。 当在 select 表达式中使用 group-by 子句时，在投影或 order-by 子句中只存在分组表达式名称、聚合或常量表达式。

以下示例计算每种产品的平均订购数量：

```
select avg(d.Quantity) from OrderDetails as d
group by d.ProductId
```

在 SELECT 表达式中，可以在没有显式 group-by 子句的情况下使用基于组的聚合。在这种情况下，会将所有元素视为单个组。这等效于基于常量指定分组的情形。例如，请看下面的表达式：

```
select avg(d.Quantity) from OrderDetails as d
```

2. 数学函数

A	E	函数	说明
Y	Y	**Abs** (value)	返回 value 的绝对值。
Y	Y	**Ceiling** (value)	返回不小于 value 的最小整数。
Y	Y	**Floor** (value)	返回不大于 value 的最大整数。
Y	Y	**Power** (value, expression)	返回对指定 value 求指定的 expression 幂次所得的结果。
Y	Y	**Round** (value)	返回 value 的整数部份，舍入到最近的整数。
Y	Y	**Round** (value, digits)	返回 value，舍入到最近的指定的 digits。
Y	Y	**Truncate** (value, digits)	返回 value，截断至最近的指定 digits。

1）Abs

返回 value 的绝对值。

参数

value：Int16、Int32、Int64、Byte、Single、Double 和 Decimal。

返回值

value 的类型。

2）Ceiling

返回不小于 value 的最小整数。

value：Single、Double 和 Decimal

返回值

value 的类型。

参数

3）**Floor**

返回不大于 value 的最大整数。

参数

value：Single、Double 和 Decimal

返回值

value 的类型

4）**Power**

参数

返回对指定 value 求指定的 expression 幂次所得的结果。

返值值

value：Int32、Int64、Double 或 Decimal。

exponent：Int64、Double 或 Decimal。

返回值

value 的类型

5）**Round**

参数

返回 value，舍入到最近的整数或的指定 digits。

参数

value：Double 或 Decimal。

digits：Int16 或 Int32。

返回值

value 的类型。

示例

```
Round(748.58)
Round(748.58, 1)
```

6）**Truncate**

参数

返回 value，截断至最近的指定 digits。

3. 字符串函数

A	E	函数	说明
Y	Y	**Contact** (string1, string2)	返回包含追加了 string1 的 string2 的字符串。
Y	Y	**Contains** (string, target)	如果 target 包含在 string 中，则返回 true。
Y	Y	**EndsWith** (string, target)	如果 target 以 string 结尾，则返回 true。
Y	Y	**IndexOf** (target, string)	返回 target 在 string 中的位置，如果没找到则返回 0。 返回 1 指示 string 的起始位置。 索引号从 1 开始。
Y	Y	**Left** (string, length)	返回 string 左侧开始的前 length 个字符。 如果 string 的长度小于 length，则返回整个字符串。
Y	Y	**Length** (string)	返回字符串的 (Int32) 长度，以字符为单位。

Y	Y	**LTrim** (string)	返回没有前导空格的 string。
Y	Y	**Replace** (string1, string2, string3)	返回 string1，其中所有 string2 都替换为 string3。
N	Y	**Reverse** (string)	
Y	Y	**Right** (string, leng th)	返回 string 的后 length 个字符。 如果 string 的长度小于 length，则返回整个字符串。
Y	Y	**RTrim** (string)	返回没有尾随空格的 string。
Y	Y	**Substring** (string, start, length)	返回字符串的从 start 位置开始、长度为 length 个字符的子字符串。 start 为 1 指示字符串的第一个字符。 索引号从 1 开始。
Y	Y	**StartsWith** (string, target)	如果 string 以 target 开头，则返回 true。
Y	Y	**ToLower** (string)	返回全部大写字符都转换为小写字符的 string。
Y	Y	**ToUpper** (string)	返回全部小写字符都转换为大写字符的 string。
Y	Y	**Trim** (string)	返回没有前导空格和尾随空格的 string。

1）**Contact**

返回包含追加了 string1 的 string2 的字符串。

参数

string1：将 string2 追加到其后的字符串。

String2：追加到 string1 之后的字符串。

返回值

一个 String。

示例

```
Concat('abc','123')
```

2）**Contains**

如果 target 包含在 string 中，则返回 true。

参数

string：在其中进行搜索的字符串。

target：所搜索的目标字符串。

返回值

如果 target 包含在 string 中，则为 true；否则为 false。

示例

```
Contains('abc','bc')
```

3）**EndsWith**

如果 target 以 string 结尾，则返回 true。

参数

string：在其中进行搜索的字符串。

target：在 string 末尾搜索的目标字符串。

返回值

如果 string 以 target 结尾，则返回 True；否则返回 false。

示例

```
-- The following example returns true.
EndsWith('abc', 'bc')
```

4）**IndexOf**

返回 target 在 string 中的位置，如果没找到则返回 0。 返回 1 指示 string 的起始位置。 索引号从 1 开始。

参数

target：要搜索的字符串。
string：在其中进行搜索的字符串。

返回值

Int32 。

示例

```
-- The following example returns 4.
IndexOf('xyz', 'abcxyz')
```

5）**Left**

返回 string 左侧开始的前 length 个字符。 如果 string 的长度小于 length，则返回整个字符串。

参数

string：String。
length：Int16、Int32、Int64 或 Byte。 length 不能小于零。

返回值

一个 String。

示例

```
-- The following example returns abc.
Left('abcxyz', 3)
```

6）**Length**

返回字符串的 (Int32) 长度，以字符为单位。

参数

string：String。

返回值

Int32

示例

```
-- The following example returns 6.
Legth('abcxyz')
```

7）**LTrim**

返回没有前导空格的 string。

参数

String

返回值

一个 String。

示例

```
-- The following example returns abc.
LTrim(' abc')
```

8）**Replace**

返回 string1，其中所有 string2 都替换为 string3。

参数

一个 String。

返回值

一个 String。

示例

```
-- The following example returns abcxyz.
Concat('abcdef', 'def', 'xyz')
```

9）**Right**

返回 string 的后 length 个字符。 如果 string 的长度小于 length，则返回整个字符串。

参数

string：String。

length：Int16、Int32、Int64 或 Byte。 length 不能小于零。

返回值

一个 String。

示例

```
-- The following example returns xyz.
Right('abcxyz', 3)
```

10）**RTrim**

返回没有尾随空格的 string。

参数

expression：一个 String。

返回值

一个 String。

11）**Substring**

返回字符串的从 start 位置开始、长度为 length 个字符的子字符串。 start 为 1 指示字符串的第一个字符。 索引号从 1 开始。

参数

string：String。

start：Int16、Int32、Int64 和 Byte。 start 不能小于一。

length：Int16、Int32、Int64 和 Byte。 length 不能小于零。

返回值

一个 String。

示例

```
-- The following example returns xyz.
Substring('abcxyz', 4, 3)
```

12）**StartsWith**

如果 string 以 target 开头，则返回 true。

参数

string：在其中进行搜索的字符串。

target：在 string 开头搜索的目标字符串。

返回值

如果 string 以 target 开头，则返回 True；否则返回 false。

示例

```
-- The following example returns true.
StartsWith('abc', 'ab')
```

13）**ToLower**

返回全部大写字符都转换为小写字符的 string。

参数

expression：一个 String。

返回值

一个 String。

示例

```
-- The following example returns abc.
ToLower('ABC')
```

14）**ToUpper**

返回全部小写字符都转换为大写字符的 string。

参数

expression：一个 String。

返回值

一个 String。

示例

```
-- The following example returns ABC.
ToUpper('abc')
```

15）**Trim**

返回没有前导空格和尾随空格的 string。

参数

一个 String。

返回值

一个 String。

示例

```
-- The following example returns abc.
Trim(' abc ')
```

4. 日期和时间函数

A	E	函数	说明
N	**Y**	**AddNanoseconds** (expression, number)	
N	**Y**	**AddMicroseconds** (expression, number)	
N	**Y**	**AddMilliseconds** (expression, number)	
N	**Y**	**AddSeconds** (expression, number)	
N	**Y**	**AddMinutes** (expression, number)	
N	**Y**	**AddHours** (expression, number)	

N	Y	**AddDays** (expression, number)	
N	Y	**AddMonths** (expression, number0	
N	Y	**AddYears** (expression, number)	
N	Y	**CreateDateTime** (year, month, day, hour, minute, second)	
N	Y	**CreateDateTimeOffset** (year, month, day, hour, minute, second, tzoffset)	
N	Y	**CreateTime** (hour, minute, second)	
N	Y	**CurrentDateTime** ()	
N	Y	**CurrentDateTimeOffset** ()	
N	Y	**CurrentUtcDateTime** ()	
Y	Y	**Day** (expression)	将 expression 的日期部分作为介于 1 到 31 之间的 Int32 返回。
Y	Y	**DayOfYear** (expression)	将 expression 的日部分作为一个介于 1 到 366 之间的 Int32 返回。其中，对于闰年的最后一天将返回 366。
N	Y	**DiffNanoseconds** (startExpression, endExpression)	
N	Y	**DiffMilliseconds** (startExpression, endExpression)	
N	Y	**DiffMicroseconds** (startExpression, endExpression)	
N	Y	**DiffSeconds** (startExpression, endExpression)	
N	Y	**DiffMinutes** (startExpression, endExpression)	
N	Y	**DiffHours** (startExpression, endExpression)	
N	Y	**DiffDays** (startExpression, endExpression)	
N	Y	**DiffMonths** (startExpression, endExpression)	
N	Y	**DiffYears** (startExpression, endExpression)	
N	Y	**GetTotalOffsetMinutes** (datetimeoffset)	
Y	Y	**Hour** (expression)	将 expression 的小时部分作为一个介于 0 到 23 之间的 Int32 返回。
N	Y	**Millisecond** (expression)	
Y	Y	**Minute** (expression)	将 expression 的分钟部分作为一个介于 0 到 23 之间的 Int32 返回。
Y	Y	**Month** (expression)	将 expression 的月部分作为一个介于 1 到 12 之间的 Int32 返回。
Y	Y	**Second** (expression)	将 expression 的秒部分作为一个介于 0 到 59 之间的 Int32 返回。
N	Y	**TruncateTime** (expression)	
Y	Y	**Year** (expression)	将 expression 的年部分作为 Int32 返回。

1）**Day**

将 expression 的日期部分作为介于 1 到 31 之间的 Int32 返回。

参数

expression：DateTime 或 String

返回值

Int

示例

Day(#2012-12-9#)

Day('2012-12-9')

2）**DayOfYear**

将 expression 的日部分作为一个介于 1 到 366 之间的 Int32 返回。其中，对于闰年的最后一天将返回 366。

参数

expression：DateTime 或 String

返回值

Int32

示例

DayOfYear(#2012-10-7#)

DayOfYear('2012-10-7')

3）**Hour**

将 expression 的小时部分作为一个介于 0 到 23 之间的 Int32 返回。

参数

exression：DateTime 或 String

返回值

Int32

示例

Hour(#2012-10-7 10:12:32#)

4）**Minute**

将 expression 的分钟部分作为一个介于 0 到 23 之间的 Int32 返回。

参数

exression：DateTime 或 String

返回值

Int32

示例

Minute(#2012-10-7 10:12:32#)

5）**Month**

将 expression 的月部分作为一个介于 1 到 12 之间的 Int32 返回。

参数

exression：DateTime 或 String

返回值 Int32

示例

Month(#2012-10-7#)

6）**Second**

参数

exression： DateTime 或 String

返回值

Int

示例

Second(#2012-10-7 10:12:32#)

7）**Year**

将 expression 的年部分作为 Int32 返回。

参数

DateTime 或 String

返回值

Int32

示例

Year(#2012-10-7#)

5. 按位规范函数

A	E	函数	说明
Y	Y	**BitWiseAnd** (value1, value2)	按照 value1 和 value2 的类型返回 value1 和 value2 的位与结果。
Y	Y	**BitWiseNot** (value)	返回 value 的位求反结果。
Y	Y	**BitWiseOr** (value1, value2)	按照 value1 和 value2 的类型返回 value1 和 value2 的位或结果。
Y	Y	**BitWiseXor** (value1, value2)	按照 value1 和 value2 的类型返回 value1 和 value2 的位异或结果。

1）**BitWiseAnd**

参数

整数类型

示例

BitWiseAnd(1,3)

2）**BitWiseNot**

参数

整数类型

示例

BitWiseNot(1)

3）**BitWiseOr**

参数

整数类型

示例

4）**BitWiseXor**

参数

整数类型

示例

BitWiseXor(1,3)

6. 其它函数

A	E	函数	说明
Y	Y	**NewGuid** ()	
Y	N	**IIF**(expr, value1, value2)	按照表达式的值，返回两个值中的一个。

1）NewGuid

示例

```
var esql = "select newguid() as Guid, e.FirstName from Employees as e";
var q = db.CreateQuery<IDataRecord>(esql);
foreach (var item in q)
    Console.WriteLine("{0} {1}", item[0], item[1]);
```

2）IIF

按照表达式的值，返回两个值中的一个。

参数

expr 布尔值

用来判断真伪的表达式。

value1

expr 为 True 返回的值。

value2

expr 为 False 返回的值。

返回值

value1 的类型

示例

```
var esql = "select value iif(e.Country = 'Chinese', 'CN', 'Other') from Employees as e";
var q = db.CreateQuery(esql);
```

三. 比较运算符

A	E	运算符	说明
Y	Y	=（等于）	比较两个表达式是否相等。
Y	Y	>（大于）	比较两个表达式以确定左侧表达式的值是否大于右侧表达式的值。
Y	Y	>=（大于或等于）	比较两个表达式以确定左侧表达式的值是否大于或等于右侧表达式的值。
Y	Y	IS [NOT] NULL	确定查询表达式是否为 null。
Y	Y	<（小于）	比较两个表达式以确定左侧表达式的值是否小于右侧表达式的值。
Y	Y	<=（小于或等于）	比较两个表达式以确定左侧表达式的值是否小于或等于右侧表达式的值。
Y	Y	[NOT] BETWEEN	确定表达式的结果值是否在指定范围内。
Y	Y	!=（不等于）	比较两个表达式以确定左侧表达式是否不等于右侧表达式。
Y	Y	[NOT] LIKE	确定特定字符串是否与指定的模式匹配。

1. =（等于）

比较两个表达式是否相等。

语法

```
expression=expression
```

```
or
expression==expression
```

参数

expression
任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式等于右侧表达式，则为 true；否则为 false。

注释

== 运算符等效于 =。

2. >（大于）

比较两个表达式以确定左侧表达式的值是否大于右侧表达式的值。

语法

```
expression > expression
```

参数

expression
任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式的值大于右侧表达式的值，则为 true；否则为 false。

3. >=（大于或等于）

比较两个表达式以确定左侧表达式的值是否大于或等于右侧表达式的值。

语法

```
expression>=expression
```

参数

expression
任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式的值大于或等于右侧表达式的值，则为 true；否则为 false。

4. IS [NOT] NULL

确定查询表达式是否为 null。

语法

```
expression IS [ NOT ] NULL
```

参数

expression

任何有效的查询表达式。 不可以是集合，不可含有集合成员，也不可以是具有集合类型属性的记录类型。

NOT

对 IS NULL 的 EDM.Boolean 结果取反。

返回值

如果 expression 返回 null，则为 true；否则为 false。

注释

使用 IS NULL 可确定外部联接的元素是否为 null：

```
select c
from Customers as c
    left join Orders as o on c.CustomerID = o.CustomerID
where o is not null
```

使用 IS NULL 可确定成员是否有实际值：

```
select p from Products as p where p.Category not is null
```

5. <（小于）

比较两个表达式以确定左侧表达式的值是否小于右侧表达式的值。

语法

```
expression<expression
```

参数

expression

任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式的值小于右侧表达式的值，则为 true；否则为 false。

6. <=（小于或等于）

比较两个表达式以确定左侧表达式的值是否小于或等于右侧表达式的值。

语法

```
expression<=expression
```

参数

expression

任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式的值小于或等于右侧表达式的值，则为 true；否则为 false。

7. [NOT] BETWEEN

确定表达式的结果值是否在指定范围内。

语法

```
expression [ NOT ] BETWEEN begin_expression AND end_expression
```

参数

expression：要测试是否在 begin_expression 和 end_expression 所定义的范围内的任何有效表达式。

begin_expression：任何有效表达式。

end_expression：任何有效表达式。

NOT：指定对 BETWEEN 的结果取反。

AND：用作一个占位符，指示 expression 应该处于由 begin_expression 和 end_expression 指定的范围内。

返回值

如果 expression 处于由 begin_expression 和 end_expression 指定的范围内，则为 true；否则为 false。

注释

若要指定某个排除范围，请使用大于 (>) 和小于 (<) 运算符而不要使用 BETWEEN。

示例

```
select p from Products as p where p.ProductId Between 10 and 100
select p from Products as p where p.ProductId not between 10 and 100
```

8. !=（不等于）

比较两个表达式以确定左侧表达式是否不等于右侧表达式。 !=（不等于）运算符在功能上等效于 <> 运算符。

语法
expression!=expression or expression <> expression

参数

expression：任何有效表达式。 两个表达式都必须具有可隐式转换的数据类型。

结果类型

如果左侧表达式不等于右侧表达式，则为 true；否则为 false。

9. [NOT] LIKE

确定特定字符 String 是否与指定模式相匹配。

语法

```
match [NOT] LIKE pattern [ESCAPE escape]
```

参数

match：计算结果为 String 的 实体 SQL 表达式。

pattern：要与指定 String 匹配的模式。

escape：一个转义符。

NOT：指定对 LIKE 的结果取反。

返回值

如果 string 与模式相匹配，则为 true；否则为 false。

四. 逻辑和 Case 表达式运算符

A	E	运算符	说明
Y	Y	&&	逻辑与。
Y	Y	!	逻辑非。
Y	Y	||	逻辑或。
Y	Y	CASE	求出一组布尔表达式的值以确定结果。
Y	Y	THEN	当 WHEN 子句取值为 true 时的结果。

1. &&(AND)

如果两个表达式均为 true，则返回 true；否则，返回 false 或 NULL。

语法

```
boolean_expression AND boolean_expression
or
boolean_expression && boolean_expression
```

参数

boolean_expression
返回布尔值的任何有效表达式。
注意：符号“&&”与 AND 运算符具有相同的功能。

2. ！(NOT)

对 Boolean 表达式求反。

语法

```
NOT boolean_expression
or
! boolean_expression  boolean_expression
```

参数

布尔值的任何有效表达式。

注释

符号“!”与 NOT 运算符具有相同的功能。

3. ||(OR)

组合两个 Boolean 表达式。

语法

```
boolean_expression OR boolean_expression
or
boolean_expression || boolean_expression
```

参数

boolean_expression
布尔值的任何有效表达式。

注释

OR 是 实体 SQL 逻辑运算符。它用于组合两个条件。在一个语句中使用多个逻辑运算符时，首先计算 AND 运算符，然后计算 OR 运算符。不过，使用括号可以更改求值的顺序。
双竖线 (||) 与 OR 运算符具有相同的功能。

4. CASE

求出一组 Boolean 表达式的值以确定结果。

语法

```
CASE
        WHEN Boolean_expression THEN result_expression
```

```
    [ ...n ]
     [
    ELSE else_result_expression
     ]
END
```

参数

n：一个占位符，表明可以使用多个 WHEN Boolean_expression THEN result_expression 子句。

THEN result_expression：作为在 Boolean_expression 的计算结果为 true 时返回的表达式。result expression 是任何有效的表达式。

ELSE else_result_expression：比较运算的结果都不为 true 时返回的表达式。如果忽略此参数且比较运算计算的结果不为 true，CASE 将返回空值。else_result_expression 是任何有效的表达式。else_result_expression 及任何 result_expression 的数据类型必须相同或必须是隐式转换的数据类型。

WHEN Boolean_expression：使用 CASE 搜索格式时所计算的 Boolean 表达式。Boolean_expression 是任何有效的 Boolean 表达式。

返回值

从 result_expression 和可选 else_result_expression 的类型集中返回优先级最高的类型。

示例

```
select value
       (case
              when e.Country = 'Chinese' then 'CN',
              when e.Country = 'English' then 'EN',
              when e.Country = 'HongKong' then 'HK',
              else 'other'
        end)
from Employees as e
```

5. THEN

当 WHEN 子句取值为 true 时的结果。

```
WHEN when_expression THEN then_expression
```

参数

when_expression：任何有效的 Boolean 表达式。

then_expression：任何返回集合的有效查询表达式。

注释

如果 when_expression 取值为 true，则结果为对应的 then-expression。如果任何 WHEN 条件均未得以满足，将求出 else-expression 的值。然而，如果没有 else-expression，则结果为空值。

五. 查询运算符

A	E	运算符	说明
Y	Y	FROM	指定 SELECT 语句中使用的集合。
Y	Y	GROUP BY	指定由查询 (SELECT) 表达式返回的对象要分入的组。
Y	Y	GroupPartition	返回从聚合与之相关的组分区提取的参数值集合。
Y	Y	HAVING	指定组或聚合的搜索条件。
Y	Y	LIMIT	

Y	Y	ORDER BY	指定用于 SELECT 语句所返回的对象的排序顺序。
Y	Y	SKIP	
Y	Y	TOP	指定查询结果中将只返回第一组行
Y	Y	WHERE	按条件筛选由查询返回的数据

1. FROM 查询

语法

```
FROM expression [ ,...n ] as C
```

参数

expression

任何可生成集合以用作 SELECT 语句中的源的有效查询表达式。

注释

FROM 子句是一个或多个 FROM 子句项的逗号分隔列表。FROM 子句可用来为 SELECT 语句指定一个或多个数据源。

1）单个数据源

FROM 子句的最简单形式是单个查询表达式，标识用作 SELECT 语句中的源的集合和别名，如下例所示：

```
var esql = @"select p from Products as p";
var q = db.CreateQuery<Product>(esql);
```

其中 Proudcts 中 db 的属性。除了例用属性，你还可以使用类名作为数据源。例如：

```
var esql = @"select p from NorthwinDemo.Product as p";
var q = db.CreateQuery<Product>(esql);
```

其中 Product 为实体类，而 NorthwindDemo 为实体类 Product 的命名空间。由于 db 的类名为 NorthwindDemo.NorthwindDataContext，命名空间与实体类的命名空间相同，因此，可以简写为：

```
var esql = @"select p from Product as p";
var q = db.CreateQuery<Product>(esql);
```

2）多个数据源

```
var esql = @"select p from Product as p, Orders as o";
var q = db.CreateQuery<Product>(esql);
```

3）内联接

INNER JOIN 生成两个集合的约束笛卡儿积，

语法：FROM C AS c [INNER] JOIN D AS d ON e（其中 [INNER] 表示 INNER 关键字可以省略）。

示例一

```
var esql = @"select o.OrderId, d.UnitPrice
             from Orders as o Inner Join
             OrderDetails as d on o.OrderId == d.OrderId";
var q = db.CreateQuery(esql);
```

示例二

```
var esql = @"select o.OrderId, d.ProductId, p.UnitPrice
                from Orders as o
                    Inner Join OrderDetails as d on o.OrderId == d.OrderId
                    Inner Join Products as p on d.ProductId == p.ProductId";
```

```
var q = db.CreateQuery(esql);
```

4）左连接

语法：FROM C AS c LEFT JOIN D AS d ON e

示例一

```
var esql = @"select o.OrderId, d.UnitPrice
             from Orders as o
                  left join OrderDetails as d on o.OrderId == d.OrderId";
var q = db.CreateQuery(esql);
```

示例二

```
var esql = @"select o.OrderId
             from Orders as o
                  left join OrderDetails as d on o.OrderId == d.OrderId
                  left join Products as p on d.ProductId == p.ProductId";
var q = db.CreateQuery(esql);
```

示例三

```
var esql = @"select o.OrderId, c.CategoryId, p.ProductId
             from Orders as o
                  left join OrderDetails as d on o.OrderId == d.OrderId
                  left join Products as p on d.ProductId == p.ProductId
                  left join Categories as c on p.CategoryId == c.CategoryId";
var q = db.CreateQuery(esql);
```

5）嵌套查询

示例

```
var esql = @"select od.o.OrderID as OrderID
               from ( select o, d
                      from Orders as o
                      inner join OrderDetails as d on o.OrderID = d.OrderID ) as od";
var q = db.CreateQuery(esql);
```

2. GROUP BY 查询

指定由查询 (SELECT) 表达式返回的对象要分入的组。

语法

```
[ GROUP BY aliasedExpression [ ,...n ] ]
```

参数

aliasedExpression

要对其执行分组的任何有效查询表达式。expression 可以是属性或者是引用 FROM 子句所返回的属性的非聚合表达式。GROUP BY 子句中的每一个表达式的求值结果必须为可比较相等性的类型。这些类型通常为标量基元类型，如数字、字符串和日期。不可按集合分组。

注释

指定 GROUP BY 子句时，无论是显式指定还是隐式指定（例如，通过查询中的 HAVING 子句指定），当前作用域都将隐藏，并且将引入新的作用域。

SELECT 子句、HAVING 子句和 ORDER BY 子句将无法再引用 FROM 子句中指定的元素名。您只能引用分组表达式本身。为此，可以为每个分组表达式指定新的名称（别名）。如果没有为分组表达式指定别名，实体 SQL 将尝试使用别名生成规则来生成别名，如下例中所示。

```
SELECT g1, g2, ...gn FROM c as c1
GROUP BY e1 as g1, e2 as g2, ...en as gn
```

GROUP BY 子句中的表达式无法引用先前在相同 GROUP BY 子句中定义的名称。

示例一

```
var esql = "select SID from Products as p group by p.SupplierID as SID";
var q = db.CreateQuery<IDataRecord>(esql);
```

示例二

```
var esql = "select SupplierID from Products as p group by p.SupplierID";
var q = db.CreateQuery<IDataRecord>(esql);
```

除了分组名称外，还可以在 SELECT 子句、HAVING 子句和 ORDER BY 子句中指定聚合。聚合中包含的表达式针对组中的每一个元素进行求值。聚合运算符缩减所有这些表达式的值（通常，但非总是，缩减为单个值）。聚合表达式可以引用父作用域中可见的原始元素名，或者引用 GROUP BY 子句本身引入的任何新名称。虽然聚合出现在 SELECT 子句、HAVING 子句和 ORDER BY 子句中，但是实际上它们是在与分组表达式相同的作用域中求值的，如下面的示例中所示。

示例一

```
var esql = @"select ProductId, sum(o.Quantity) as SumQuantity
             from OrderDetails as o
             group by o.ProductId";
var q = db.CreateQuery(esql);
```

示例二

```
var esql = @"select CategoryID, SupplierID, Count()
            from Products as p
            group by p.CategoryID, p.SupplierID";
var db.CreateQuery<IDataRecord>(esql).Execute();
```

示例三

```
select OrderID
from ( select o, d
       from Orders as o
            inner join OrderDetails as d on o.OrderID = d.OrderID ) as od
group by od.o.OrderID
```

3. GroupPartition 查询

返回从聚合与之相关的当前组分区提取的参数值集合。GroupPartition 聚合是基于组的聚合，没有基于集合的形式。

语法

```
GROUPPARTITION( [ALL|DISTINCT] expression )
```

参数

expression

任何 实体 SQL 表达式。

注释

示例一

下面的查询，将产品按 CategoryId 进行分组，同时获取分组的产品。

```
var esql = @"select CategoryID, GroupPartition(p) as Products
               from Products as p
               group by p.CategoryID";
var items = db.CreateQuery<IDataRecord>(esql);
foreach (var item in items)
{
    foreach (Product p in (IEnumerable<Product>)item["Products"])
    {
        Console.WriteLine(p.ProductName);
    }
}
```

示例二

下面的查询，将产品按 CategoryId 进行分组，并计算每组产品的 UnitPrice 的总和。

```
var esql = @"select CategoryID, sum(GroupPartition(p.UnitPrice)) as PriceSum
             from Products as p
             group by p.CategoryID";
var items = db.CreateQuery<IDataRecord>(esql);
foreach (var item in items)
{
    Console.WriteLine(item["PriceSum"]);
}
```

下面的 ESQL 以上面的语句语义上是相同的。但是建议采用上面的语句，因为它生成的 SQL 语句只有一条，具有更好的效率。

```
select CategoryID, GroupPartition(sum(p.UnitPrice)) as PriceSum
from Products as p
group by p.CategoryID
```

4. HAVING 查询

指定组或聚合的搜索条件。

语法

```
[ HAVING search_condition ]
```

参数

search_condition

指定组或聚合应满足的搜索条件。

注释

HAVING 子句用于对分组结果指定附加筛选条件。HAVING 子句与 WHERE 子句的工作方式类似，只是它应用在 GROUP BY 操作之后。这意味着 HAVING 子句只能引用分组别名和聚合，如下面的示例所示。

示例一

```
select value c from Products as p
group by p.CategoryId as c
having c > 1000
```

示例二

```
select c from Products as p
```

```
group by p.CategoryId as c, p.SupplierId
having c > 1000
```

示例三

```
select value c from Products as p
group by p.CategoryId as c
having max(p.CategoryId) > 1000
```

5. LIMIT 查询

在 SELECT 子句中使用 LIMIT 子句可执行物理分页。LIMIT 子句应放置在 SELECT 子句的最后面。

语法

```
[ LIMIT n ]
or
[TAKE n]
```

参数

n：将选择的项的数量。

注意：LIMIT 与 TAKE 是等效的，并且与 TOP 不能同时出现在同一条 SELECT 查询子句中。

示例

```
select e from Employees as e limit 10
```

或

```
select e from Employees as e take 10
```

6. ORDER BY 查询

指定用于 SELECT 语句所返回的对象的排序顺序。

语法

```
[ ORDER BY    {order_by_expression    [ ASC | DESC ] }  [ ,...n ]  ]
```

参数

order_by_expression

指定作为排序依据的属性的任何有效查询表达式。可以指定多个排序表达式。ORDER BY 子句中排序表达式的顺序将决定排序后结果集的结构。

ASC

指定所指定属性中的值应按升序排序，即从最低值往最高值排序。这是默认选项。

DESC

指定所指定属性中的值应按降序排序，即从最高值往最低值排序。

注释

ORDER BY 子句在逻辑上应用于 SELECT 子句的结果。ORDER BY 子句可以使用选择列表中各项的别名来引用这些项。ORDER BY 子句也可引用当前处于作用域内的其他变量。

ORDER BY 子句中的每个表达式的计算结果都必须是可按顺序比较是否不等（小于或大于，等等）的某一类型。这些类型通常为标量基元类型，如数字、字符串和日期。

示例一

```
select e from Employees as e order by e.FirstName asc
```

与下面的语句等效

```
select e from Employees as e order by e.FirstName
```

示例二

```
select e from Employees as e order by e.FirstName desc
```

7. SKIP 查询

在 SELECT 子句中使用 SKIP 谓词可执行物理分页

语法

```
[ SKIP n ]
```

参数

n：要跳过的项数。

示例一

```
select p from Products as p skip 10
```

示例二

```
select p from Products as p skip 10 limit 10
```

8. TOP 查询

指定查询结果中最多返回的记录数。

语法

```
[ TOP (n) ]
```

参数

n：一个数值表达式，指定要返回的行数。n 可以是单个数值或单个参数。

注释

TOP 表达式必须是单个数值或单个参数，并且大于 0。

示例一

```
select top(10) p from Products as p
```

示例二

```
select top(@topNum) p from Products as p
```

9. WHERE 查询

按条件筛选由查询返回的数据，WHERE 子句直接应用在 FROM 子句之后。

语法

```
[ WHERE expression ]
```

参数

expression

Boolean 类型。

注释

WHERE 子句的语义与针对 Transact-SQL 所述的语义相同。 它将源集合的元素限定为传递条件的元素，以此限制查询表达式所生成的对象。WHERE 子句直接应用在 FROM 子句之后，任何分组、排序或投影操作之前。FROM 子句中定义的所有元素名称对 WHERE 子句的表达式都是可见的。

示例

```
select p from Products as p where p.UnitPrice > 10
```

六. 引用运算符

A	E	函数	说明
N	Y	CREATEREF	
N	Y	DEREF	
N	Y	NAVIGATE	
N	Y	REF	

七. 集合运算符

A	E	运算符	说明
N	Y	ANYELEMENT	
Y	Y	EXCEPT	返回由 EXCEPT 操作数左侧的查询表达式返回而不由 EXCEPT 操作数右侧的查询表达式返回的任何非重复值的集合。
Y	Y	[NOT] EXISTS	确定集合是否为空。
N	Y	FLATTEN	
Y	Y	[NOT] IN	
Y	Y	INTERSECT	返回 INTERSECT 操作数左右两边的两个查询表达式均返回的所有非重复值的集合。
N	Y	OVERLAPS	
N	Y	SET	
Y	Y	UNION	将两个或更多查询的结果组合成单个集合。

1. EXCEPT

返回由 EXCEPT 操作数左侧的查询表达式返回而不由 EXCEPT 操作数右侧的查询表达式返回的任何非重复值的集合。 所有表达式都必须与 expression 一样属于同一类型或属于公共基类型或派生类型。

语法

```
expression EXCEPT expression
```

参数

expression

返回一个集合以与从其他查询表达式返回的集合进行比较的任何有效查询表达式。

返回值

与 expression 具有相同类型或属于公共基类型或派生类型的一个集合。

EXCEPT 是 实体 SQL 集运算符之一。 所有 实体 SQL 集运算符都是从左到右进行求值。 下表显示 实体 SQL 集运算符的优先级。

优先级	运算符
最高	UNION
	EXCEPT
最低	EXISTS

示例一

```
(select p1.ProductName from Products as p1 where p1.CategoryID = 2)
except
(select p2.ProductName from Products as p2 where p2.CategoryID = 3)
```

示例二

```
(select p1 from Products as p1 where p1.CategoryID = 2)
except
(select p2 from Products as p2 where p2.CategoryID = 3)
```

2. EXISTS

确定集合是否为空。

语法

```
[NOT] EXISTS (expression)
```

参数

expression

返回集合的任何有效的表达式。

NOT

指定对 EXISTS 的结果取反。

返回值

如果集合不为空，则为 true；否则为 false。

注释

EXISTS 是 实体 SQL 集运算符之一。 所有 实体 SQL 集运算符都是从左到右进行求值

示例

```
select p from Products as p
where exists(select p from Products as p where p.UnitPrice > 0)
```

3. IN

确定某个值是否与某个集合中的任何值匹配。

语法

```
value [ NOT ] IN expression
```

参数

value：返回匹配值的任何有效表达式。

[NOT]：指定对 IN 的 Boolean 结果取反。

expression

返回集合以测试是否具有匹配的任何有效表达式。 所有表达式都必须与 value 一样属于同一类型或属于公共基类型或派生类型。

返回值

如果在集合中找到此值，则为 true；如果值为空或集合为空，则为空；否则为 false。 使用 NOT IN 可对 IN 的结果取反。

示例一

```
select o from Orders as o where o.OrderId in { 1, 2 ,3 }
```

示例二

```
select o from Orders as o
where o.OrderId in ( select value d.OrderId from OrderDetails as d )
```

4. INTERSECT

返回 INTERSECT 操作数左右两边的两个查询表达式均返回的所有非重复值的集合。

语法

```
expression INTERSECT expression
```

参数

expression：返回一个集合以与从其他查询表达式返回的集合进行比较的任何有效查询表达式。

返回值

与 expression 具有相同类型或属于公共基类型或派生类型的一个集合。

示例

```
var esql = @"(select p.ProductId, p.ProductName from Products as p where p.UnitPrice < 100 )
                intersect
                (select p.ProductId, p.ProductName from Products as p where p.UnitPrice > 200)";
var q = db.CreateQuery<IDataRecord>(esql);
```

5. UNION

将两个或更多查询的结果组合成单个集合。

语法

```
expression UNION expression
```

参数

expression：返回一个集合以与该集合进行组合的任何有效查询表达式。所有表达式都必须与 expression 一样属于同一类型或属于公共基类型或派生类型。

UNION：指定组合多个集合并将其作为单个集合返回。

返回值

与 expression 具有相同类型或属于公共基类型或派生类型的一个集合。

注释

UNION 是 实体 SQL 集运算符之一。 所有 实体 SQL 集运算符都是从左到右进行求值。

示例

```
(select p.ProductID, p.ProductName from Products as p where p.UnitPrice < 100 )
union
(select p.ProductID, p.ProductName from Products as p where p.UnitPrice > 200)
```

6. MULTISET

根据值列表创建多集的实例。 MULTISET 构造函数中的所有值都必须具有兼容类型 T。 不允许使用空的多集构造函数。

语法

```
MULTISET (expression [{, expression }] )

or

{ expression [{, expression }] }
```

参数

expression：任何有效的值列表。

返回值

类型为 MULTISET<T> 的集合。

注释

多集构造函数根据值列表创建多集的实例。 该构造函数中的所有值都必须具有兼容类型。

例如，下面的表达式创建整数的多集。

MULTISET(1, 2, 3)

{1, 2, 3}

示例一

```
select o from Orders as o where o.OrderId in { 1, 2 ,3 }
```

示例二

```
select o from Orders as o where o.OrderId in MultiSet( 1, 2 ,3 )
```

7. ROW

从一个或多个值构造结构上类型化的匿名记录。

语法

```
ROW (expression [ AS alias ] [,...] )
```

参数

expression

任何有效的查询表达式，该表达式返回要在行类型中构造的值。

alias

为在行类型中指定的值指定别名。 如果未提供别名，则 实体 SQL 会尝试基于 实体 SQL 别名生成规则来生成别名。

返回值

一个行类型。

注释

使用 ESQL 中的行构造函数可以从一个或多个值构造结构上类型化的匿名记录。 行构造函数的结果类型为行类型，其字段类型对应于用于构造该行的值的类型。 例如，下面的表达式构造一个类型为 DataRow(a int, b string, c int) 的值。

```
ROW(1 AS a,  "abc" AS b,  a+34 AS c)
```

如果没有为行构造函数中的表达式提供别名，则系统会尝试生成一个别名。

以下规则适用于在行构造函数中指定表达式别名：

1、行构造函数中的表达式不能引用同一构造函数中的其他别名。

2、同一行构造函数中的两个表达式不能具有相同别名。

示例

```
select row(p.CategoryId, p.UnitPrice) from Products as p
```

8. TREAT

将特定基类型的对象视为指定派生类型的对象。

语法

```
TREAT (expression as type)
```

参数

expression：任何返回实体的有效查询表达式。

type：一个实体类型。

返回值

一个具有指定数据类型的值。

注释

TREAT 用于在相关类之间执行向上转换。 例如，如果 Employee 派生自 Person 且 p 的类型为 Person，则 TREAT(p AS Employee) 会将泛型 Person 实例向上转换为 Employee；即，使您可以将 p 视为 Employee。

TREAT 用于可以执行类似于以下查询的继承方案：

```
SELECT TREAT(p AS Employee)
FROM ContainerName.Person AS p
WHERE p IS OF (Employee)
```

此查询将 Person 实体向上转换为 Employee 类型。如果 p 的值的实际类型不是 Employee，则表达式会生成值 null。

八. 类型运算符

A	E	运算符	说明
Y	Y	CAST	将一种数据类型的表达式转换为另一种数据类型的表达式。
N	Y	COLLECTION	
Y	Y	IS [NOT] OF	确定表达式的类型是否为指定类型或指定类型的某个子类型。
Y	Y	OFTYPE	从查询表达式返回特定类型的对象集合。
Y	Y	命名类型构造函数	用于创建实体类型或复杂类型的实例。
Y	Y	MULTISET	根据值列表创建多集的实例。
Y	Y	ROW	从一个或多个值构造结构上类型化的匿名记录。
Y	Y	TREAT	将特定基类型的对象视为指定派生类型的对象。

1. CAST

将一种数据类型的表达式转换为另一种数据类型的表达式。

语法

```
CAST ( expression AS data_type)
```

参数

expression：任何可转换为 data_type 的有效表达式。

data_type：系统提供的目标数据类型，类型为公共语言运行库 (CLR) 类型。

返回值

返回与 data_type 相同的值。

注释

强制转换表达式的语义与 Transact-SQL CONVERT 表达式类似。 强制转换表达式用于将一种类型的值转换为另一种类型的值。

```
CAST( e as T )
```

如果 e 具有某种类型 S，且 S 可转换为 T，则上面的表达式是有效的强制转换表达式。T 必须为基元（标量）类型。使用强制转换表达式视为显式转换。显式转换可能截断数据或丧失精度。

示例

```
select value cast(c.CategoryId as string) from Categories as c
select value cast(c.CategoryId as short) from Categories as c
select value cast(c.CategoryId as long) from Categories as c
```

2. IS [NOT] OF

确定表达式的类型是否为指定类型或指定类型的某个子类型。

语法

```
expression IS [ NOT ] OF ( [ONLY] type)
```

参数

expression：要确定其类型的任何有效查询表达式。

NOT：对 IS OF 的 EDM.Boolean 结果取反。

ONLY：指定仅当 expression 的类型为 type，而不是其任何子类型时，IS OF 才返回 true。

type：要针对其测试 expression 的类型。

返回值

如果 expression 的类型为 T 且 T 为基类型或 type 的派生类型，则返回 true；如果 expression 在运行时为 null，则返回 null；否则返回 false。

注释

表达式 expression IS NOT OF (type) 和 expression IS NOT OF (ONLY type) 在语法上分别等效于 NOT (expression IS OF (type)) 和 NOT (expression IS OF (ONLY type))。

示例一

```
var esql = "select c from Contacts as c where c is of EmployeeContact";
var q = db.CreateQuery<EmployeeContact>(esql);
```

示例二

```
var esql = "select c from Contacts as c where c is not of EmployeeContact";
var q = db.CreateQuery<EmployeeContact>(esql);
```

示例三

```
var esql = "select c from Contacts as c where c is of only EmployeeContact";
var q = db.CreateQuery<EmployeeContact>(esql);
```

3. OFTYPE

从查询表达式返回特定类型的对象集合。

语法

```
OFTYPE (expression, [ONLY] test_type)
```

参数

expression：返回对象集合的任何有效的查询表达式。

test_type：要对 expression 返回的每个对象进行测试的类型。该类型必须由命名空间进行限定。

返回值

test_type 类型或 test_type 的基类型或派生类型的对象集合。如果指定 ONLY，则仅返回 test_type 的实例或空集合。

示例一

```
var esql = "select c from oftype(Contacts, FullContact) as c";
var q = db.CreateQuery<EmployeeContact>(esql);
```

示例二

```
string esql = "select c from oftype(Contacts, only FullContact) as c";
var q = db.CreateQuery<EmployeeContact>(esql);
```

4. 命名类型构造函数

语法

```
[{identifier. }] identifier( [expression [{, expression }][{, expression as identifier}]] )
```

参数

identifier：作为简单标识符或带引号的标识符的值。

expression：类型构造函数的参数或类型的属性。

构造函数参数

在 ALinq Dynamic 的语句中，为特定的类型创建实例，是不需要 new 關鍵字的。

例句：在下面的例中，创建一个 System.DateTime 的實例。

```
var esql = "System.DateTime(2012,11,5)";
var result = db.CreateQuery<DateTime>(esql).Single();
```

1）初始化属性值

在创建实例的时候，同时还可以为实例的属性赋值

例句：下面的例子，创建一个 NorthwindDemo.Person 的实例，同时还将 FisrtName 属性赋值为 mike ，LastName 属性赋值为 mak ，值得注意的时，赋值语的值在前，接着是 AS 关键字，最后是属性名。

```
var employeeId = 12345678;
var esql = "NorthwindDemo.Person(123, 'mike' as FirstName, 'mak' as LastName)";
var person = db.CreateQuery<NorthwindDemo.Person>(esql, employeeId).Single();
```

其等于下面的语句

```
var person = new Person(123);
person.FirstName = "mike";
person.LastName = "mak";
```

2）使用预定义的类

在 ALinq Dynamic 中，预定义了一些常用的类型（具体请看下表），对于预定义的类型，可以直接用 ALinq Dynamic 中的名称。

例如：

```
var esql = "System.DateTime(2012,11,5)"
```

等效于：

```
var esql = "DateTime(2012,11,5)"
```

ALinq Dynamic 中的名称	.Net 框架中的类
DateTime	System.DateTime
TimeSpan	System.TimeSpan
Guid	System.Guid
Math	System.Math
Converter	System.Converter
object	System.Object
bool	System.Boolean
char	System.Char
string	System.String
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
float	System.Single
double	System.Double

decimal	System.Decimal
DateTime	System.DateTime
TimeSpan	System.TimeSpan
Guid	System.Guid
Math	System.Math
Convert	System.Convert
Binary	System.Byte[]
X	System.Byte[]

5. MULTISET

根据值列表创建多集的实例。

语法

```
MULTISET (expression [{, expression }] )
or
{ expression [{, expression }] }
```

参数

expression：任何有效的值列表。

返回值

类型为 T[]（数组） 的集合。

示例

```
select value m from MULTISET(1, 2, 3) as m
select value m from {1, 2, 3} as m
select o from Orders as o where o.OrderId in { 1, 2 ,3 }
```

6. ROW

从一个或多个值构造结构上类型化的匿名记录。

语法

```
ROW (expression [ AS alias ] [,...] )
```

参数

expression：任何有效的查询表达式，该表达式返回要在行类型中构造的值。

alias：为在行类型中指定的值指定别名。

返回值

一个行类型。

示例一

```
var esql = @"Row('mike' as FirstName, 'mak' as LastName,
        Row('ansiboy@163.com' as Email,
            '13434126607' as Phone) as Contact)";

var row = db.CreateQuery<IDataRecord>(esql).Single();
Console.WriteLine("{0} {1}", row["FirstName"], row["LastName"]);
Console.WriteLine("{0} {1}", ((IDataRecord)row["Contact"])["Email"], ((IDataRecord)row["Contact"])["Phone"]);
```

示例二

```
select row(p.CategoryId, p.UnitPrice) from Products as p
```

```
group by p.CategoryId, p.UnitPrice
having max(p.UnitPrice) > 1000
```

7. TREAT

将特定基类型的对象视为指定派生类型的对象。

语法

```
TREAT (expression as type)
```

参数

expression：任何返回实体的有效查询表达式。

注意：指定表达式的类型必须为指定数据类型的子类型，或者该数据类型必须为表达式的类型的子类型。

type：一个实体类型。

注意：指定表达式必须为指定数据类型的子类型，或者该数据类型必须为该表达式的子类型。

返回值

一个具有指定数据类型的值。

示例

```
var esql = @"select treat(c as FullContact) from Contacts as c where c is of FullContact";
var q = db.CreateQuery<FullContact>(esql);
```

九. 其它运算符

A	E	运算符	说明
Y	Y	+（字符串串连）	
Y	Y	.（成员访问）	
Y	Y	--（注释）	
N	Y	FUNCTION	

1. +（字符串串联）

连接两个字符串。

语法

```
expression + expression
```

参数

expression：字符串类型的任何有效表达式。 或者能够隐式转换为字符串类型的表达式。

示例

```
select value e.FirstName + ' ' + e.LastName as Name from Employees as e
```

2. 成员访问

点运算符 (.) 是 实体 SQL 成员访问运算符。 使用成员访问运算符可生成结构化概念模型类型实例的属性或字段的值。

语法

```
expression.identifier[([arg1,[arg2]])]
```

参数

expression：类的实例。

identifier：对象实例的属性或字段。

注释

点 (.) 运算符可以用于从记录中提取字段，类似于提取复杂类型或实体类型的属性。

示例一

Employee 是 Order 实体类的一个成员， o.Employee.FirstName 将 Employee 实体类的 FisrtName 成员提取出来。

```
select o.Employee.FirstName from Orders as o
```

示例二

下面的语句，在选择 ProductName 属性的同时，调用 Trime 方法去掉 ProductName 的前后空格。

```
select value p.ProductName.Trim() from Products as p
```

3. --（注释）

实体 SQL 查询可以包含注释。 注释行以两个短划线 (--) 开头。

语法

```
-- text_of_comment
```

参数

text_of_comment：包含注释文本的字符串。

示例

```
select p from Products as p    -- add a comment here
```

十. 扩展查询方法

方法名称	Entity SQL 语句	说明
GroupBy	Group By	按指定的条件对查询结果进行分组。
OrderBy	OrderBy	按指定条件对查询结果进行排序。
Select	Select	将查询结果限制为仅包含在指定投影中定义的属性。
Skip	Skip	按指定条件对查询结果进行排序并跳过指定数目的结果。
Take	Take	将查询结果限制为指定的项数。
Where	Where	将查询限制为包含与指定筛选条件匹配的结果。

1. GroupBy

按指定的条件对查询结果进行分组。

语法

```
public IQueryable<IDataRecord> GroupBy(
    string keys,
    string projection,
    params object[] parameters
)
```

参数

keys：作为结果分组依据的键列。

projection：用于定义投影的所选属性的列表。

parameters：此方法中使用的零个或多个参数。

返回值

一个新的 IQueryable<IDataRecord> 实例，等效于应用了 Group By 的原始实例。

示例

```
var q = db.Products.GroupBy("CategoryId", "CategoryId, count()").Execute();
```

```
foreach (var item in q)
{
    Console.WriteLine("{0} {1}", item[0], item[1]);
}
```

2. OrdeyBy

按指定条件对查询结果进行排序。

语法

```
public ObjectQuery<T> OrderBy(
    string keys,
    params object[] parameters
)
```

参数

keys：作为结果排序依据的键列。

parameters：此方法中使用的零个或多个参数。

返回值

一个新的 IQueryable<T> 实例，等效于应用了 ORDER BY 的原始实例。

示例

```
db.GetTable<Product>().OrderBy("it.ProductName")
db.GetTable<Product>().OrderBy("it.ProductName desc")
db.GetTable<Product>().OrderBy("it.ProductName asc")
```

3. Select

将查询结果限制为仅包含在指定投影中定义的属性。

语法

```
public IQueryable<IDataRecord> Select(
    string projection,
    params object[] parameters
)
```

参数

projection：用于定义投影的所选属性的列表。

parameters：此方法中使用的零个或多个参数。

返回值

类型：System.Linq.IQueryable<IDataRecord>，一个 IDataRecord 类型的新 IQueryable<T> 实例，等效于应用了 SELECT 的原始实例。

示例

```
var q = db.GetTable<Product>().Select("it.ProductName, it.UnitsInStock, it.UnitsOnOrder");
foreach (var item in q)
{
    Console.WriteLine("{0} {1} {2}", item[1], item[2], item[3]);
}
```

4. Skip

按指定条件对查询结果进行排序并跳过指定数目的结果。

语法

```
public IQueryable<T> Skip(
    string keys,
    string count,
    params object[] parameters
)
```

参数

keys：作为结果排序依据的键列。

count：要跳过的结果数。它可以是常量或参数引用。

parameters：在分析时应在作用域内的一组可选查询参数。

返回值

一个新 IQueryable<T> 实例，等效于同时应用了 SKIP 的原始实例。

示例

```
db.GetTable<Product>().Skip("10");
db.GetTable<Product>().Skip("@0, 10);
db.GetTable<Product>().Skip("@skip", new ObjectParameter("skip", 10));
```

5. Take

将查询结果限制为指定的项数。

语法

```
public IQueryable<T> Take(
    string count,
    params object[] parameters
)
```

参数

count：字符串形式的结果项数。

parameters：在分析时应在作用域内的一组可选查询参数。

返回值

一个新的 IQueryable<T> 实例，等效于应用了 TAKE 或 LIMIT 的原始实例。

示例

```
db.GetTable<Product>().Take("10")
db.GetTable<Product>().Take("@0", 10)
db.GetTable<Product>().Take("@take", new ObjectParameter("take", 10))
```

6. Where

将查询限制为包含与指定筛选条件匹配的结果。

语法

```
public IQueryable<T> Where(
    string predicate,
    params Object[] parameters
)
```

参数

predicate：筛选谓词。

parameters：此方法中使用的零个或多个参数。

返回

一个新的 IQueryable<T> 实例，等效于应用了 WHERE 的原始实例。

示例一

```
db.Customers.Where("it.Region == 'WA'")
```

示例二

```
var customerId_Set = new[] { "AROUT", "BOLId", "FISSA" };
db.Orders.Where("it.CustomerId in @0", new[] { customerId_Set });
```

V、附录

1.数据上下文与实体类代码

文档中的示例代码，你可以在 ALinq Dynamic 的单元测试项目中找到。但为了阅读上方便，在这里给出常用类的定义（为了减少篇幅，有所删减，不能直接运行）。

1）NorthwindDataContext 类的定义

文本中的 db 变量，就是 NorthwindDataContext 类的实例。

```
namespace NorthwindDemo
{
    [ALinq.Mapping.ProviderAttribute(typeof(ALinq.SQLite.SQLiteProvider))]
    public partial class NorthwindDataContext : ALinq.DataContext
    {
        public ALinq.Table<Category> Categories
        {
            get { return this.GetTable<Category>(); }
        }
        public ALinq.Table<CustomerCustomerDemo> CustomerCustomerDemos
        {
            get { return this.GetTable<CustomerCustomerDemo>(); }
        }
        public ALinq.Table<CustomerDemographic> CustomerDemographics
        {
            get { return this.GetTable<CustomerDemographic>(); }
        }
        public ALinq.Table<Customer> Customers
        {
            get { return this.GetTable<Customer>(); }
        }
        public ALinq.Table<Employee> Employees
        {
            get { return this.GetTable<Employee>(); }
        }
        public ALinq.Table<EmployeeTerritory> EmployeeTerritories
        {
            get { return this.GetTable<EmployeeTerritory>(); }
        }
        public ALinq.Table<OrderDetail> OrderDetails
        {
            get { return this.GetTable<OrderDetail>(); }
        }
```

```
        public ALinq.Table<Order> Orders
        {
            get { return this.GetTable<Order>(); }
        }
        public ALinq.Table<Product> Products
        {
            get { return this.GetTable<Product>(); }
        }
        public ALinq.Table<Region> Regions
        {
            get { return this.GetTable<Region>(); }
        }
        public ALinq.Table<Shipper> Shippers
        {
            get { return this.GetTable<Shipper>(); }
        }
        public ALinq.Table<Supplier> Suppliers
        {
            get { return this.GetTable<Supplier>(); }
        }
        public ALinq.Table<Territory> Territories
        {
            get { return this.GetTable<Territory>(); }
        }
    }
}
```

2）IEmployee 接口

```
namespace NorthwindDemo
{
    public interface IEmployee
    {
        int EmployeeId { get; set; }

        string LastName { get; set; }

        string FirstName { get; set; }

        string Title { get; set; }

        string TitleOfCourtesy { get; set; }

        DateTime? BirthDate { get; set; }

        DateTime? HireDate { get; set; }

        string Address { get; set; }
```

```
        string City { get; set; }

        string Region { get; set; }

        string PostalCode { get; set; }

        string Country { get; set; }

        string HomePhone { get; set; }

        string Extension { get; set; }

        byte[] Photo { get; set; }

        string Notes { get; set; }

        int? ReportsTo { get; set; }

        string PhotoPath { get; set; }
    }
}
```

3）Employee 实体类

```
namespace NorthwindDemo
{
    [Table(Name="Employees")]
    public partial class Employee : INotifyPropertyChanging, INotifyPropertyChanged, IEmployee
    {
        [Column(AutoSync=AutoSync.OnInsert, CanBeNull=false, IsPrimaryKey=true, IsDbGenerated=true,
UpdateCheck=UpdateCheck.Never)]
        public int EmployeeId
        {
            get;
            set;
        }

        [Column(DbType="VarChar(20)", CanBeNull=false, UpdateCheck=UpdateCheck.Never)]
        public string LastName
        {
            get;
            set;
        }

        [Column(DbType="VarChar(10)", CanBeNull=false, UpdateCheck=UpdateCheck.Never)]
        public string FirstName
        {
```

```
        get;set;
    }

    [Column(DbType="VarChar(30)", UpdateCheck=UpdateCheck.Never)]
    public string Title
    {
        get;set;
    }

    [Column(DbType="VarChar(25)", UpdateCheck=UpdateCheck.Never)]
    public string TitleOfCourtesy
    {
        get;set;
    }

    [Column(UpdateCheck=UpdateCheck.Never)]
    public System.Nullable<System.DateTime> BirthDate
    {
        get;set;
    }

    [Column(UpdateCheck=UpdateCheck.Never)]
    public System.Nullable<System.DateTime> HireDate
    {
        get;set;
    }

    [Column(DbType="VarChar(60)", UpdateCheck=UpdateCheck.Never)]
    public string Address
    {
        get;set;
    }

    [Column(DbType="VarChar(15)", UpdateCheck=UpdateCheck.Never)]
    public string City
    {
        get;set;
    }

    [Column(DbType="VarChar(15)", UpdateCheck=UpdateCheck.Never)]
    public string Region
    {
        get;set;
    }

    [Column(DbType="VarChar(10)", UpdateCheck=UpdateCheck.Never)]
```

```
public string PostalCode
{
    get;set;
}

[Column(DbType="VarChar(15)", UpdateCheck=UpdateCheck.Never)]
public string Country
{
    get;set;
}

[Column(DbType="VarChar(24)", UpdateCheck=UpdateCheck.Never)]
public string HomePhone
{
    get;set;
}

[Column(DbType="VarChar(4)", UpdateCheck=UpdateCheck.Never)]
public string Extension
{
    get;set;
}

[Column(UpdateCheck=UpdateCheck.Never)]
public byte[] Photo
{
    get;set;
}

[Column(UpdateCheck=UpdateCheck.Never)]
public string Notes
{
    get;set;
}

[Column(UpdateCheck=UpdateCheck.Never)]
public System.Nullable<int> ReportsTo
{
    get;set;
}

[Column(DbType="VarChar(255)", UpdateCheck=UpdateCheck.Never)]
public string PhotoPath
{
    get;set;
}
```

```
        [Association(Name="Employee_Employee", ThisKey="EmployeeId", OtherKey="ReportsTo")]
        public EntitySet<Employee> Employees
        {
            get;set;
        }

        [Association(Name="Employee_EmployeeTerritory", Storage="_EmployeeTerritories", ThisKey="EmployeeId",
OtherKey="EmployeeId")]
        public EntitySet<EmployeeTerritory> EmployeeTerritories
        {
            get;set;
        }

        [Association(Name="Employee_Order", Storage="_Orders", ThisKey="EmployeeId", OtherKey="EmployeeId")]
        public EntitySet<Order> Orders
        {
            get;set;
        }

        [Association(Name="Employee_Employee", Storage="_ReportsToEmployee", ThisKey="ReportsTo",
OtherKey="EmployeeId", IsForeignKey=true)]
        public Employee ReportsToEmployee
        {
            get;set;
        }
    }
}
```