LINQ：语言集成查询，从数据源检索信息的功能。

可由LINQ使用的数据源必须实现IEnumerable接口，IEnumerable<T>泛型接口或非泛型的IEnumerable接口。

所有的C#数组都实现了IEnumerable<T>接口。

例：程序linq\_test1

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ的简单使用

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test1

{

class LINQTest

{

static void Main(string[] args)

{

int[] nums = { 1, -2, 3, 0, -4, 5};

// 创建LINQ查询，但不产生结果

var positive\_nums = from n in nums

where n > 0 select n;

*Console*.*Write*("The positive vlaues in nums: ");

// 执行查询

foreach (int i in positive\_nums)

*Console*.*Write*(i + " ");

*Console*.*WriteLine*();

*Console*.*WriteLine*("Setting nums[1] to 99");

nums[1] = 99;

*Console*.*Write*("The positive values in nums after change: ");

// 再一次执行查询

foreach (int i in positive\_nums)

*Console*.*Write*(i + " ");

*Console*.*WriteLine*();

}

}

}

输出结果为：

The positive vlaues in nums: 1 3 5

Setting nums[1] to 99

The positive values in nums after change: 1 99 3 5

从输出结果看出，数据源改变，查询结果也会跟着变化。

查询中的数据类型的关联方式：

var positive\_nums = from n in nums

where n > 0 select n;

范围变量为n，其类型必须与数据源的元素类型一致。数据源实现IEnumerable<T>接口，就可以自动推断范围变量的类型。如果数据源实现非泛型的IEnumerable接口，则需要显示指定范围变量的类型：如

var positive\_nums = from int n in nums

查询变量为positive\_nums，其类型必须是IEnumerable<T>的实例，T的值由select语句指定的值的类型确定，在本例中，T为int，这是因为范围变量n的类型为int，通常查询变量声明为var。

通过foreach执行查询时，迭代变量的类型必须与select子句指定的类型相同，即与范围变量的类型相同，可以使用var来声明迭代变量。

LINQ查询必须从from关键字开始，以select或group结束。

where子句：

例：程序linq\_test2

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ查询

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test2

{

class LINQTest

{

static void Main(string[] args)

{

string[] strs = { ".com", ".net", "hsNameA.com", "hsNameB.net",

"test", ".network", "hsNameC.net", "hsNameD.com"};

var net\_addrs = from addr in strs

where addr.*Length* > 4 && addr.*EndsWith*(".net", *StringComparison*.*Ordinal*)

select addr;

// 执行查询的迭代变量为var

foreach (var str in net\_addrs)

*Console*.*WriteLine*(str);

}

}

}

使用orderby

子句排序结果

orderby sort-on how

其中how必须是ascending（升序）或descending（降序）。默认是升序。sort-on可以是数据源中的整个元素，或元素的单个字段。

可以使用oderby语句进行多组排序：

orderby sort-onA direction, sort-onB direction, sort-onC direction

在这种形式中，sort-onA执行主要排序的标准，然后按照sort-onB排序每组相等的项，再sort-onC排序之后相等的项。每一个sort都指定排序的进一步标准。

例：程序linq\_test3

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ查询：多组orderby排序

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test3

{

class Account

{

public string FirstName { get; private set; }

public string LastName { get; private set; }

public double Balance { get; private set; }

public string AccountNumber { get; private set; }

public Account(string fn, string ln, double bal, string accnum)

{

FirstName = fn;

LastName = ln;

Balance = bal;

AccountNumber = accnum;

}

}

class LINQTest

{

static void Main(string[] args)

{

// 对象初始化器

Account[] accounts = { new Account("Tom", "Smith", 100.23, "132CK"),

new Account("Tom", "Smith", 1000.23, "132CD"),

new Account("Ralph", "Jones", 1923.85, "436CD"),

new Account("Ralph", "Jones", 987.12, "436MM"),

new Account("Ted", "Krammer", 3223.19, "897CD"),

new Account("Ralph", "Jones", -123.32, "434CK"),

new Account("Sara", "Smith", 5017.40, "543MM"),

new Account("Sara", "Smith", 3495.79, "547CD"),

};

var account\_info = from acc in accounts

orderby acc.LastName, acc.FirstName, acc.Balance

select acc;

*Console*.*WriteLine*("Accounts in sorted order: ");

string temp = "";

// 执行查询

foreach (Account acc in account\_info)

{

if (temp != acc.FirstName)

{

*Console*.*WriteLine*();

temp = acc.FirstName;

}

*Console*.*WriteLine*("{0}, {1}\tAcc#: {2}, {3, 10:C}",

acc.LastName, acc.FirstName,

acc.AccountNumber, acc.Balance);

}

}

}

}

输出结果为：

Accounts in sorted order:

Jones, Ralph Acc#: 434CK, ￥-123.32

Jones, Ralph Acc#: 436MM, ￥987.12

Jones, Ralph Acc#: 436CD, ￥1,923.85

Krammer, Ted Acc#: 897CD, ￥3,223.19

Smith, Sara Acc#: 547CD, ￥3,495.79

Smith, Sara Acc#: 543MM, ￥5,017.40

Smith, Tom Acc#: 132CK, ￥100.23

Smith, Tom Acc#: 132CD, ￥1,000.23

select子句

投影：select子句可以返回变量的特定部分、对范围变量应用一些操作或变换的结果或者根据从范围变量检索的部分信息构造的新对象类型。

例：程序linq\_test4

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ:select子句

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test4

{

class EmailAddress

{

public string Name { get; set; }

public string Address { get; set; }

public EmailAddress(string n, string a)

{

Name = n;

Address = a;

}

}

class LINQTest

{

static void Main(string[] args)

{

double[] nums = { -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5};

// 数组中大于0的项的平方根

// select子句对范围变量进行操作

var sqrt\_roots = from n in nums

where n > 0

select *Math*.*Sqrt*(n);

*Console*.*WriteLine*("The square roots of the positive values"

+ " rounded to two decimal places: ");

// 执行查询

foreach (double r in sqrt\_roots)

*Console*.*WriteLine*("{0:#.##}", r);

EmailAddress[] addrs = {

new EmailAddress("Herb", "Herb@HerbSchildt.com"),

new EmailAddress("Tom", "Tom@HerbSchildt.com"),

new EmailAddress("Sara", "Sara@Herbschildt.com")

};

// select子句返回范围变量的部分信息

var email\_addrs = from entry in addrs

select entry.Address;

*Console*.*WriteLine*("The email addresses are");

// 执行查询

foreach (var s in email\_addrs)

*Console*.*WriteLine*(" " + s);

}

}

}

输出结果：

The square roots of the positive values rounded to two decimal places:

4.05

3.48

10.04

5.02

The email addresses are

Herb@HerbSchildt.com

Tom@HerbSchildt.com

Sara@Herbschildt.com

select子句返回一个新的类型对象

例：程序linq\_test5

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ查询：select子句

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test5

{

class ContactInfo

{

public string Name { get; set; }

public string Email { get; set; }

public string Phone { get; set; }

public ContactInfo(string n, string a, string p)

{

Name = n;

Email = a;

Phone = p;

}

}

class EmailAddress

{

public string Name { get; set; }

public string Address { get; set; }

public EmailAddress(string n, string a)

{

Name = n;

Address = a;

}

}

class LINQTest

{

static void Main(string[] args)

{

ContactInfo[] contacts = {

new ContactInfo("Herb", "Herb@HerbSchildt.com", "555-1010"),

new ContactInfo("Tom", "Tom@HerbSchildt.com", "555-1101"),

new ContactInfo("Sara", "Sara@HerbSchildt.com", "555-0101")

};

// select子句返回一个新的类型对象

var email\_list = from entry in contacts

select new EmailAddress(entry.Name, entry.Email);

*Console*.*WriteLine*("The email list is");

foreach (EmailAddress e in email\_list)

*Console*.*WriteLine*("{0}: {1}", e.Name, e.Address);

}

}

}

输出结果：

The email list is

Herb: Herb@HerbSchildt.com

Tom: Tom@HerbSchildt.com

Sara: Sara@HerbSchildt.com

嵌套的from子句：常用于遍历包含在数据源中的另一个数据源。

例：程序linq\_test6

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：from子句嵌套

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test6

{

class CharPair

{

public char First;

public char Second;

public CharPair(char c, char c2)

{

First = c;

Second = c2;

}

}

class LINQTest

{

static void Main(string[] args)

{

char[] chrs = { 'A', 'B', 'C'};

char[] chrs2 = { 'X', 'Y', 'Z'};

// 嵌套的from子句

var pairs = from ch1 in chrs

from ch2 in chrs2

select new CharPair(ch1, ch2);

*Console*.*WriteLine*("All combinations of ABC with XYZ: ");

foreach (var p in pairs)

*Console*.*WriteLine*("{0} {1}", p.First, p.Second);

}

}

}

输出结果：

All combinations of ABC with XYZ:

A X

A Y

A Z

B X

B Y

B Z

C X

C Y

C Z

group子句：group子句可以结束查询，select子句也可以结束查询。

group range-variable by key

group子句的结果是包含IGrouping<TKey, TElement>元素的序列。对于返回值的查询，其中查询变量的类型是IEnumerable<IGrouping<TKey, TElement> >

例：程序linq\_test7

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ查询：group子句

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test7

{

class LINQTest

{

static void Main(string[] args)

{

string[] web\_sites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",

"hsNameD.com", "hsNameE.org", "hsNameF.org",

"hsNameG.tv", "hsNameH.net", "hsNameI.tv"};

// 查询变量web\_addrs是IEnumerable<IGrouping<string, string> >

*IEnumerable*<*IGrouping*<string, string> > web\_addrs = from addr in web\_sites

where addr.*LastIndexOf*(".") != -1

group addr by addr.*Substring*(addr.*LastIndexOf*("."));

// 执行查询

foreach (*IGrouping*<string, string> sites in web\_addrs)

{

*Console*.*WriteLine*("Web sites grouped by " + sites.*Key*);

foreach (string site in sites)

*Console*.*WriteLine*(" " + site);

*Console*.*WriteLine*();

}

}

}

}

输出变量：

Web sites grouped by .com

hsNameA.com

hsNameD.com

Web sites grouped by .net

hsNameB.net

hsNameC.net

hsNameH.net

Web sites grouped by .org

hsNameE.org

hsNameF.org

Web sites grouped by .tv

hsNameG.tv

hsNameI.tv

使用into子句创建查询延续：使用into子句生成临时结果，以便后续的查询使用该临时结果产生最终结果。

into name query-body

其中，name是遍历临时结果的范围变量，由query-body指定的后续查询使用该范围变量。

例：程序linq\_test8

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：into子句，延续查询

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test8

{

class LINQTest

{

static void Main(string[] args)

{

string[] web\_sites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",

"hsNameD.com", "hsNameE.org", "hsNameF.org",

"hsNameG.tv", "hsNameH.net", "hsNameI.tv"};

// into子句，ws为后续查询的范围变量

var web\_addrs = from addr in web\_sites

where addr.*LastIndexOf*(".") != -1

group addr by addr.*Substring*(addr.*LastIndexOf*(".")) into ws

where ws.*Count*() > 2

select ws;

// 执行查询

foreach (var sites in web\_addrs)

{

*Console*.*WriteLine*("Web sites grouped by " + sites.*Key*);

foreach (var site in sites)

*Console*.*WriteLine*(" " + site);

*Console*.*WriteLine*();

}

}

}

}

输出结果：

Web sites grouped by .net

hsNameB.net

hsNameC.net

hsNameH.net

let子句：在查询中创建变量，用于后续的操作

例：程序linq\_test9

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：let子句

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test9

{

class LINQTest

{

static void Main(string[] args)

{

string[] strs = { "alpha", "beta", "gamma"};

// 使用let创建一个临时变量，存储嵌套from

// 查询的数据源

// 将strs字符串数组对应的字符数组按升序排列

var chrs = from str in strs

let char\_arry = str.*ToCharArray*()

from ch in char\_arry

orderby ch

select ch;

*Console*.*WriteLine*("The individual characters in sorted order:");

foreach (char c in chrs)

*Console*.*Write*(c + " ");

*Console*.*WriteLine*();

}

}

}

输出结果：

The individual characters in sorted order:

a a a a a b e g h l m m p t

使用join子句连接两个序列

from range-varA in data-sourceA

join range-varB in data-sourceB

on range-varA.property equals range-varB.property

data-sourceA和data-sourceB必须有可以用于比较的共有数据。通过on指定部分指定比较的项。

例：程序linq\_test10

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：join子句连接序列

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test10

{

class Item

{

public string Name { get; set; }

public int ItemNumber { get; set; }

public Item(string n, int num)

{

Name = n;

ItemNumber = num;

}

}

class InStockStatus

{

public int ItemNumber { get; set; }

public bool InStock { get; set; }

public InStockStatus(int n, bool b)

{

ItemNumber = n;

InStock = b;

}

}

class Temp

{

public string Name { get; set; }

public bool InStock { get; set; }

public Temp(string n, bool b)

{

Name = n;

InStock = b;

}

}

class LINQTest

{

static void Main(string[] args)

{

Item[] items = {

new Item("Pliers", 1424),

new Item("Hammer", 7892),

new Item("Wrench", 8534),

new Item("Saw", 6411)

};

InStockStatus[] status = {

new InStockStatus(1424, true),

new InStockStatus(7892, false),

new InStockStatus(8534, true),

new InStockStatus(6411, true)

};

// join子句连接两个序列

var stock\_list = from item in items

join entry in status

on item.ItemNumber equals entry.ItemNumber

select new Temp(item.Name, entry.InStock);

*Console*.*WriteLine*("Item\tAvaiable\n");

foreach (Temp t in stock\_list)

*Console*.*WriteLine*("{0}\t{1}", t.Name, t.InStock);

}

}

}

输出结果为：

Item Avaiable

Pliers True

Hammer False

Wrench True

Saw True

匿名类型：

创建匿名类型：

new {nameA = valueA, nameB = valueB,…}

其中，名称指定转换为只读属性的标识符。

new { Count = 10, Max = 100, Min = 0}

创建具有3个公有只读属性的匿名类类型。初始化匿名类型，使用对象初始化器（无法显式调用构造函数）。

例：程序linq\_test11

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：匿名类型

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test11

{

class Item

{

public string Name { get; set; }

public int ItemNumber { get; set; }

public Item(string n, int num)

{

Name = n;

ItemNumber = num;

}

}

class InStockStatus

{

public int ItemNumber { get; set; }

public bool InStock { get; set; }

public InStockStatus(int n, bool b)

{

ItemNumber = n;

InStock = b;

}

}

class LINQTest

{

static void Main(string[] args)

{

Item[] items = {

new Item("Pliers", 1424),

new Item("Hammer", 7892),

new Item("Wrench", 8534),

new Item("Saw", 6411)

};

InStockStatus[] status = {

new InStockStatus(1424, true),

new InStockStatus(7892, false),

new InStockStatus(8534, true),

new InStockStatus(6411, true)

};

// join子句连接两个序列

// 匿名类型

var stock\_list = from item in items

join entry in status

on item.ItemNumber equals entry.ItemNumber

select new { Name = item.Name, InStock = entry.InStock };

// 下面的代码执行的效果与上面的相同

// 这是投射初始化器，item.Name和entry.InStock

// 会自动生成为匿名类型的Name和InStock

// select new { item.Name, entry.InStock };

*Console*.*WriteLine*("Item\tAvaiable\n");

// 执行查询

foreach (var t in stock\_list)

*Console*.*WriteLine*("{0}\t{1}", t.Name, t.InStock);

}

}

}

输出结果：

Item Avaiable

Pliers True

Hammer False

Wrench True

Saw True

创建组连接

使用into和join子句创建组连接，组连接创建一个序列，其中的每个条目由来自第一个序列的条目和来自第二个序列的一组所有匹配的元素组成。

例：程序linq\_test12

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：使用join和into创建组连接

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test12

{

class Transport

{

public string Name { get; set; }

public string How { get; set; }

public Transport(string n, string h)

{

Name = n;

How = h;

}

}

class LINQTest

{

static void Main(string[] args)

{

string[] travel\_types = { "Air", "Sea", "Land"};

Transport[] transports = {

new Transport("Bicycle", "Land"),

new Transport("Balloon", "Air"),

new Transport("Boat", "Sea"),

new Transport("Jet", "Air"),

new Transport("Canoe", "Sea"),

new Transport("Biplane", "Air"),

new Transport("Car", "Land"),

new Transport("Cargo Ship", "Sea"),

new Transport("Train", "Land")

};

// 创建组连接

var by\_how = from how in travel\_types

join trans in transports

on how equals trans.How

into lst

select new { How = how, Tlist = lst };

// 执行查询

foreach (var t in by\_how)

{

*Console*.*WriteLine*("{0} transportation includes:", t.How);

foreach (var m in t.Tlist)

*Console*.*WriteLine*(" " + m.Name);

*Console*.*WriteLine*();

}

}

}

}

延期执行查询和立即执行查询：

延期执行查询：查询定义一组规则，直到foreach语句执行时才会实际执行这组规则。

立即执行查询：使用产生不连续结果的某种扩展方法，如Count()，Max()等，必须在调用该方法时自动执行查询。

表达式树：只可以表示表达式Lambda，不可以表示语句Lambda。

例：程序linq\_test13

// Copyright 2016.刘珅珅

// author：刘珅珅

// LINQ：表达式树

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Linq*.*Expressions*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test13

{

class LINQTest

{

static void Main(string[] args)

{

// 声明一个Lambda表达式作为数据

*Expression*<*Func*<int, int, bool>>

IsFactorExp = (n, d) => (d != 0) ? (n % d) == 0 : false;

// 将表达式数据转换为可执行代码

*Func*<int, int, bool> IsFactor = IsFactorExp.*Compile*();

// 执行表达式

if (IsFactor(10, 5))

{

*Console*.*WriteLine*("5 is a factor of 10.");

}

if (!IsFactor(10, 7))

{

*Console*.*WriteLine*("7 is not a factor of 10.");

}

}

}

}

输出结果为：

5 is a factor of 10.

7 is not a factor of 10.

扩展方法：可以向类添加功能而无需使用普通继承机制的方式。

扩展方法必须是static方法，必须包含在静态非泛型类中。

扩展方法的第一个参数的类型确定可以对其调用该扩展方法的对象的类型。

必须通过this关键字修饰第一个参数。

static ret-type name(this invoked-on-type obj, param-list)

例：程序linq\_test14

// Copyright 2016.刘珅珅

// author：刘珅珅

// 扩展方法

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Globalization*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace linq\_test14

{

static class MyExtendMethods

{

// 可以被doube类型使用的扩展方法

public static double Reciprocal(this double v)

{

return 1.0 / v;

}

// 可以被string类型使用的扩展方法

public static string RevCase(this string str)

{

string temp = "";

foreach (char ch in str)

{

if (*Char*.*IsLower*(ch))

{

temp += *Char*.*ToUpper*(ch, *CultureInfo*.*CurrentCulture*);

}

else

{

temp += *Char*.*ToLower*(ch, *CultureInfo*.*CurrentCulture*);

}

}

return temp;

}

// 可以被double类型使用的扩展方法

public static double AbsDivideBy(this double n, double d)

{

return *Math*.*Abs*(n / d);

}

}

class LINQTest

{

static void Main(string[] args)

{

double val = 8.0;

string str = "Alpha Beta Gamma";

// 扩展方法为静态方法，但其调用却类似于实例方法

*Console*.*WriteLine*("Reciprocal of {0} is {1}", val, val.Reciprocal());

*Console*.*WriteLine*(str + " after reversing case is "

+ str.RevCase());

*Console*.*WriteLine*("Result of abs divide by -2: " + val.AbsDivideBy(-2));

}

}

}

输出结果：

Reciprocal of 8 is 0.125

Alpha Beta Gamma after reversing case is aLPHA bETA gAMMA

Result of abs divide by -2: 4