EBook Gratis

APRENDIZAJE

linq

Free unaffiliated eBook created from
Stack Overflow contributors.

#linq

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: linq

It is an unofficial and free linq ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official linq.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con linq

Observaciones

LINQ es un conjunto de características introducidas en .NET Framework versión 3.5 que cierra la brecha entre el mundo de los objetos y el mundo de los datos.

Tradicionalmente, las consultas contra los datos se expresan como cadenas simples sin verificación de tipo en tiempo de compilación o soporte de IntelliSense. Además, debe aprender un lenguaje de consulta diferente para cada tipo de fuente de datos: bases de datos SQL, documentos XML, diversos servicios web, etc. LINQ realiza una consulta en una construcción de lenguaje de primera clase en C # y Visual Basic. Usted escribe consultas contra colecciones de objetos fuertemente tipadas usando palabras clave de lenguaje y operadores familiares.

Examples

Preparar

LINQ requiere .NET 3.5 o superior (o .NET 2.0 usando LINQBridge).

Agregue una referencia a **System.Core** , si aún no se ha agregado.

En la parte superior del archivo, importe el espacio de nombres:

- DO#

```
using System;
using System.Linq;
```

- VB.NET

```
Imports System.Linq
```

Las diferentes uniones en LINQ.

En los siguientes ejemplos, usaremos los siguientes ejemplos:

```
List<Product> Products = new List<Product>()
{
  new Product()
  {
    ProductId = 1,
    Name = "Book nr 1",
    Price = 25
  },
  new Product()
  {
    ProductId = 2,
```

```
    Name = "Book nr 2",
    Price = 15
  },
  new Product()
  {
    ProductId = 3,
    Name = "Book nr 3",
    Price = 20
  },
};
List<Order> Orders = new List<Order>()
{
  new Order()
  {
    OrderId = 1,
    ProductId = 1,
  },
  new Order()
  {
    OrderId = 2,
    ProductId = 1,
  },
  new Order()
  {
    OrderId = 3,
    ProductId = 2,
  },
  new Order()
  {
    OrderId = 4,
    ProductId = NULL,
  },
};
```

UNIR INTERNAMENTE

Sintaxis de consulta

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId
              select new
              {
                o.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Sintaxis del método

```
var joined = Products.Join(Orders, p => p.ProductId,
                                   o => o.OrderId,
                                     => new
                                    {
                                      OrderId   = o.OrderId,
                                      ProductId = p.ProductId,
                                      Name      = p.Name
                                    })
                     .ToList();
```

Resultado:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" }
```

IZQUIERDA COMBINACIÓN EXTERNA

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId into g
              from lj in g.DefaultIfEmpty()
              select new
              {
                //For the empty records in lj, OrderId would be NULL
                OrderId = (int?)lj.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Resultado:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" },
{ NULL, 3, "Book nr 3" }
```

Unirse a la cruz

```
var joined = (from p in Products
              from o in Orders
              select new
              {
                o.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Resultado:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" },
{ NULL, 3, "Book nr 3" },
{ 4, NULL, NULL }
```

Unirse al grupo

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId
                into t
              select new
              {
                p.ProductId,
                p.Name,
```

```
                Orders = t
              }).ToList();
```

Las `Orders` Propiedad ahora contienen un `IEnumerable<Order>` con todas las Órdenes vinculadas.

Resultado:

```
{ 1, "Book nr 1", Orders = { 1, 2 } },
{ 2, "Book nr 2", Orders = { 3 } },
{ 3, "Book nr 3", Orders = { } },
```

Cómo unirse en múltiples condiciones

Al unirse en una sola condición, puede utilizar:

```
join o in Orders
  on p.ProductId equals o.ProductId
```

Al unir en múltiples, use:

```
join o in Orders
  on new { p.ProductId, p.CategoryId } equals new { o.ProductId, o.CategoryId }
```

Asegúrese de que ambos objetos anónimos tengan las mismas propiedades, y en VB.NET, deben estar marcados como `Key` , aunque VB.NET permite varias cláusulas de `Equals` separadas por `And` :

```
Join o In Orders
  On p.ProductId Equals o.ProductId And p.CategoryId Equals o.CategoryId
```

Sintaxis de consulta y sintaxis de método

La sintaxis de consulta y la sintaxis de método son semánticamente idénticas, pero muchas personas encuentran la sintaxis de consulta más sencilla y más fácil de leer. Digamos que necesitamos recuperar todos los artículos pares ordenados en orden ascendente de una colección de números.

DO#:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6 };

// Query syntax:
IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

// Method syntax:
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);
```

VB.NET:

```
Dim numbers() As Integer = { 0, 1, 2, 3, 4, 5, 6 }

' Query syntax: '
Dim numQuery1 = From num In numbers
                Where num Mod 2 = 0
                Select num
                Order By num

' Method syntax: '
Dim numQuery2 = numbers.where(Function(num) num Mod 2 = 0).OrderBy(Function(num) num)
```

Recuerde que algunas consultas **deben** expresarse como llamadas de método. Por ejemplo, debe usar una llamada de método para expresar una consulta que recupera el número de elementos que coinciden con una condición específica. También debe usar una llamada de método para una consulta que recupera el elemento que tiene el valor máximo en una secuencia de origen. Entonces, eso podría ser una ventaja de usar la sintaxis de los métodos para hacer que el código sea más consistente. Sin embargo, por supuesto, siempre puede aplicar el método después de una llamada de sintaxis de consulta:

DO#:

```
int maxNum =
    (from num in numbers
     where num % 2 == 0
     select num).Max();
```

VB.NET:

```
Dim maxNum =
    (From num In numbers
     Where num Mod 2 = 0
     Select num).Max();
```

Métodos LINQ, y IEnumerable vs IQueryable

Los métodos de extensión LINQ en `IEnumerable<T>` toman los métodos reales [1] , ya sean métodos anónimos:

```
//C#
Func<int,bool> fn = x => x > 3;
var list = new List<int>() {1,2,3,4,5,6};
var query = list.Where(fn);

'VB.NET
Dim fn = Function(x As Integer) x > 3
Dim list = New List From {1,2,3,4,5,6};
Dim query = list.Where(fn);
```

o métodos nombrados (métodos definidos explícitamente como parte de una clase):

```
//C#
class Program {
```

```
    bool LessThan4(int x) {
        return x < 4;
    }

    void Main() {
        var list = new List<int>() {1,2,3,4,5,6};
        var query = list.Where(LessThan4);
    }
}

'VB.NET
Class Program
    Function LessThan4(x As Integer) As Boolean
        Return x < 4
    End Function
    Sub Main
        Dim list = New List From {1,2,3,4,5,6};
        Dim query = list.Where(AddressOf LessThan4)
    End Sub
End Class
```

En teoría, es posible analizar el IL del método , averiguar qué está tratando de hacer el método y aplicar la lógica de ese método a cualquier fuente de datos subyacente, no solo a los objetos en la memoria. Pero el análisis de la IL no es para los débiles de corazón.

Afortunadamente, .NET proporciona la interfaz `IQueryable<T>` y los métodos de extensión en `System.Linq.Queryable` , para este escenario. Estos métodos de extensión tienen un árbol de expresión - una estructura de datos que representa código - en lugar de un método actual, que el proveedor de LINQ entonces puede analizar [2] y convertir a una forma más apropiada para la consulta de la fuente de datos subyacente. Por ejemplo:

```
//C#
IQueryable<Person> qry = PersonsSet();

// Since we're using a variable of type Expression<Func<Person,bool>>, the compiler
// generates an expression tree representing this code
Expression<Func<Person,bool>> expr = x => x.LastName.StartsWith("A");
// The same thing happens when we write the lambda expression directly in the call to
// Queryable.Where

qry = qry.Where(expr);


'VB.NET
Dim qry As IQueryable(Of Person) = PersonSet()

' Since we're using a variable of type Expression(Of Func(Of Person,Boolean)), the compiler
' generates an expression tree representing this code
Dim expr As Expression(Of Func(Of Person, Boolean)) = Function(x) x.LastName.StartsWith("A")
' The same thing happens when we write the lambda expression directly in the call to
' Queryable.Where

qry = qry.Where(expr)
```

Si (por ejemplo) esta consulta es contra una base de datos SQL, el proveedor podría convertir esta expresión a la siguiente declaración SQL:

```
SELECT *
FROM Persons
WHERE LastName LIKE N'A%'
```

y ejecutarlo contra la fuente de datos.

Por otro lado, si la consulta es contra una API REST, el proveedor podría convertir la misma expresión en una llamada a la API:

```
http://www.example.com/person?filtervalue=A&filtertype=startswith&fieldname=lastname
```

Hay dos ventajas principales en la personalización de una solicitud de datos basada en una expresión (en lugar de cargar toda la colección en la memoria y realizar consultas locales):

- El origen de datos subyacente a menudo puede consultar de manera más eficiente. Por ejemplo, puede haber un índice en el `LastName` . Cargar los objetos en la memoria local y consultar en la memoria pierde esa eficiencia.
- Los datos pueden ser conformados y reducidos antes de ser transferidos. En este caso, la base de datos / servicio web solo necesita devolver los datos coincidentes, a diferencia del conjunto completo de Personas disponibles desde la fuente de datos.

Notas

1. Técnicamente, no toman métodos, sino delegan instancias que apuntan a métodos . Sin embargo, esta distinción es irrelevante aquí.
2. Este es el motivo de errores como " *LINQ to Entities no reconoce el método 'System.String ToString ()', y este método no se puede traducir a una expresión de tienda* ". El proveedor de LINQ (en este caso, el proveedor de Entity Framework) no sabe cómo analizar y traducir una llamada a `ToString` a un SQL equivalente.

Lea Empezando con linq en línea: https://riptutorial.com/es/linq/topic/842/empezando-con-linq

Capítulo 2: Linq usando Take while y Skip mientras que

Introducción

Take, Skip, TakeWhile y SkipWhile se denominan operadores de partición, ya que obtienen una sección de una secuencia de entrada según lo determinado por una condición determinada. Vamos a discutir estos operadores

Examples

Toma metodo

El método de toma Lleva los elementos a una posición específica a partir del primer elemento de una secuencia. **Firma de Take:**

```
Public static IEnumerable<TSource> Take<TSource>(this IEnumerable<TSource> source,int count);
```

Ejemplo:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var TakeFirstFiveElement = numbers.Take(5);
```

Salida:

El resultado es 1,5,8,4 y 9 para obtener cinco elementos.

Omitir método

Salta elementos hasta una posición específica a partir del primer elemento en una secuencia.

Firma de Skip:

```
Public static IEnumerable Skip(this IEnumerable source,int count);
```

Ejemplo

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.Take(5);
```

Salida: El resultado es 3,6,7,2 y 0 para obtener el elemento.

TakeWhile ():

Devuelve elementos de la colección dada hasta que la condición especificada sea verdadera. Si el primer elemento no satisface la condición, devuelve una colección vacía.

Firma de TakeWhile ():

```
Public static IEnumerable <TSource> TakeWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,bool>,predicate);
```

Otra firma de sobrecarga:

```
Public static IEnumerable <TSource> TakeWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,int,bool>,predicate);
```

Ejemplo I:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.TakeWhile(n => n < 9);
```

Salida:

Volverá De los elementos 1,5,8 y 4.

Ejemplo II:

```
int[] numbers = { 1, 2, 3, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.TakeWhile((n,Index) => n < index);
```

Salida:

Se devolverá del elemento 1,2,3 y 4.

SkipWhile ()

Omite elementos en función de una condición hasta que un elemento no cumple la condición. Si el primer elemento en sí no satisface la condición, entonces omite 0 elementos y devuelve todos los elementos en la secuencia.

Firma de SkipWhile ():

```
Public static IEnumerable <TSource> SkipWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,bool>,predicate);
```

Otra firma de sobrecarga:

```
Public static IEnumerable <TSource> SkipWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,int,bool>,predicate);
```

Ejemplo I:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.SkipWhile(n => n < 9);
```

Salida:

Volverá De elemento 9,3,6,7,2 y 0.

Ejemplo II:

```
int[] numbers = { 4, 5, 8, 1, 9, 3, 6, 7, 2, 0 };
var indexed = numbers.SkipWhile((n, index) => n > index);
```

Salida:

Se devolverá del elemento 1,9,3,6,7,2 y 0.

Lea Linq usando Take while y Skip mientras que en línea:
https://riptutorial.com/es/linq/topic/10810/linq-usando-take-while-y-skip-mientras-que

Capítulo 3: Modos de ejecución de métodos: transmisión inmediata, diferida, sin transmisión diferida

Examples

Ejecución diferida vs ejecución inmediata

Algunos métodos LINQ devuelven un objeto de consulta. Este objeto no contiene los resultados de la consulta; en cambio, tiene toda la información necesaria para generar esos resultados:

```
var list = new List<int>() {1, 2, 3, 4, 5};
var query = list.Select(x => {
    Console.Write($"{x} ");
    return x;
});
```

La consulta contiene una llamada a `Console.Write` , pero no se ha `Console.Write` nada a la consola. Esto se debe a que la consulta aún no se ha ejecutado y, por lo tanto, la función pasada a `Select` nunca se ha evaluado. Esto se conoce como **ejecución diferida: la ejecución de** la consulta se retrasa hasta algún punto posterior.

Otros métodos LINQ obligan a una **ejecución inmediata** de la consulta; Estos métodos ejecutan la consulta y generan sus valores:

```
var newList = query.ToList();
```

En este punto, la función que pasó a `Select` se evaluará para cada valor en la lista original, y lo siguiente se enviará a la consola:

    1 2 3 4 5

En general, los métodos LINQ que devuelven un solo valor (como `Max` o `Count`), o que devuelven un objeto que realmente contiene los valores (como `ToList` o `ToDictionary`) se ejecutan inmediatamente.

Los métodos que devuelven un `IEnumerable<T>` o `IQueryable<T>` devuelven el objeto de consulta y permiten aplazar la ejecución hasta un punto posterior.

Si un método LINQ particular obliga a una consulta a ejecutarse inmediatamente o no, se puede encontrar en MSDN - C # , o VB.NET .

Modo de transmisión (evaluación perezosa) versus modo sin transmisión

(evaluación impaciente)

De los métodos LINQ que utilizan la ejecución diferida, algunos requieren que se evalúe un solo valor a la vez. El siguiente código:

```
var lst = new List<int>() {3, 5, 1, 2};
var streamingQuery = lst.Select(x => {
    Console.WriteLine(x);
    return x;
});
foreach (var i in streamingQuery) {
    Console.WriteLine($"foreach iteration value: {i}");
}
```

saldrá:

```
3
valor de iteración de foreach: 3
5
valor de iteración de foreach: 5
1
valor de iteración de foreach: 1
2
valor de iteración de foreach: 2
```

porque la función pasada a `Select` se evalúa en cada iteración del `foreach` . Esto se conoce como **modo de transmisión** o **evaluación perezosa** .

Otros métodos LINQ (operadores de clasificación y agrupación) requieren que se evalúen *todos* los valores, antes de que puedan devolver *cualquier* valor:

```
var nonStreamingQuery = lst.OrderBy(x => {
    Console.WriteLine(x);
    return x;
});
foreach (var i in nonStreamingQuery) {
    Console.WriteLine($"foreach iteration value: {i}");
}
```

saldrá:

```
3
5
1
2
valor de iteración de foreach: 1
valor de iteración de foreach: 2
valor de iteración de foreach: 3
valor de iteración de foreach: 5
```

En este caso, debido a que los valores se deben generar para el `foreach` en orden ascendente, primero se deben evaluar todos los elementos, para determinar cuál es el más pequeño, y cuál es el siguiente más pequeño, y así sucesivamente. Esto se conoce como modo **sin transmisión** o **evaluación impaciente** .

Ya sea que un método LINQ particular use el modo de transmisión por secuencias o no, puede encontrarse en MSDN - C # , o VB.NET .

Beneficios de la ejecución diferida - construcción de consultas

La ejecución diferida permite combinar diferentes operaciones para construir la consulta final, antes de evaluar los valores:

```
var list = new List<int>() {1,1,2,3,5,8};
var query = list.Select(x => x + 1);
```

Si ejecutamos la consulta en este punto:

```
foreach (var x in query) {
    Console.Write($"{x} ");
}
```

obtendríamos la siguiente salida:

    2 2 3 4 6 9

Pero podemos modificar la consulta agregando más operadores:

```
Console.WriteLine();
query = query.Where(x => x % 2 == 0);
query = query.Select(x => x * 10);

foreach (var x in query) {
    Console.Write($"{x} ");
}
```

Salida:

    20 20 40 60

Beneficios de la ejecución diferida - consultar datos actuales

Con la ejecución diferida, si se cambian los datos a consultar, el objeto de consulta utiliza los datos en el momento de la ejecución, no en el momento de la definición.

```
var data = new List<int>() {2, 4, 6, 8};
var query = data.Select(x => x * x);
```

Si ejecutamos la consulta en este punto con un método inmediato o `foreach` , la consulta operará

en la lista de números pares.

Sin embargo, si cambiamos los valores en la lista:

```
data.Clear();
data.AddRange(new [] {1, 3, 5, 7, 9});
```

O incluso si asignamos una nueva lista a los `data` :

```
data = new List<int>() {1, 3, 5, 7, 9};
```

y luego ejecute la consulta, la consulta operará en el nuevo valor de los `data` :

```
foreach (var x in query) {
    Console.Write($"{x} ");
}
```

y dará salida a lo siguiente:

    1 9 25 49 81

Lea Modos de ejecución de métodos: transmisión inmediata, diferida, sin transmisión diferida en línea: https://riptutorial.com/es/linq/topic/7102/modos-de-ejecucion-de-metodos--transmision-inmediata--diferida--sin-transmision-diferida

Capítulo 4: Operadores de consultas estándar

Observaciones

Las consultas de Linq se escriben utilizando los Operadores de consultas estándar (que son un conjunto de métodos de extensión que funcionan principalmente en objetos de tipo `IEnumerable<T>` e `IQueryable<T>`) o utilizando Expresiones de consulta (que en el momento de la compilación, se convierten en Operador de consultas estándar método de llamadas).

Los operadores de consultas proporcionan capacidades de consulta que incluyen filtrado, proyección, agregación, clasificación y más.

Examples

Operaciones de concatenación

La concatenación se refiere a la operación de anexar una secuencia a otra.

Concat

> Concatena dos secuencias para formar una secuencia.

Sintaxis del método

```
// Concat

var numbers1 = new int[] { 1, 2, 3 };
var numbers2 = new int[] { 4, 5, 6 };

var numbers = numbers1.Concat(numbers2);

// numbers = { 1, 2, 3, 4, 5, 6 }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de filtrado

El filtrado se refiere a las operaciones de restringir el conjunto de resultados para que contengan solo aquellos elementos que satisfacen una condición específica.

Dónde

> Selecciona valores que se basan en una función de predicado.

Sintaxis del método

```
// Where

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };

var evens = numbers.Where(n => n % 2 == 0);

// evens = { 2, 4, 6, 8 }
```

Sintaxis de consulta

```
// where

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };

var odds = from n in numbers
           where n % 2 != 0
           select n;

// odds = { 1, 3, 5, 7 }
```

De tipo

Selecciona valores, en función de su capacidad para convertirse en un tipo específico.

Sintaxis del método

```
// OfType

var numbers = new object[] { 1, "one", 2, "two", 3, "three" };

var strings = numbers.OfType<string>();

// strings = { "one", "two", "three" }
```

Sintaxis de consulta

```
// Not applicable.
```

Unir Operaciones

Una combinación de dos fuentes de datos es la asociación de objetos en una fuente de datos con objetos que comparten un atributo común en otra fuente de datos.

Unirse

Une dos secuencias basadas en funciones de selector de clave y extrae pares de valores.

Sintaxis del método

```
// Join

class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Order
{
    public string Description { get; set; }
    public int CustomerId { get; set; }
}
...

var customers = new Customer[]
{
    new Customer { Id = 1, Name = "C1" },
    new Customer { Id = 2, Name = "C2" },
    new Customer { Id = 3, Name = "C3" }
};

var orders = new Order[]
{
    new Order { Description = "O1", CustomerId = 1 },
    new Order { Description = "O2", CustomerId = 1 },
    new Order { Description = "O3", CustomerId = 2 },
    new Order { Description = "O4", CustomerId = 3 },
};

var join = customers.Join(orders, c => c.Id, o => o.CustomerId, (c, o) => c.Name + "-" +
o.Description);

// join = { "C1-O1", "C1-O2", "C2-O3", "C3-O4" }
```

Sintaxis de consulta

```
// join … in … on … equals …

var join = from c in customers
           join o in orders
           on c.Id equals o.CustomerId
           select o.Description + "-" + c.Name;

// join = { "O1-C1", "O2-C1", "O3-C2", "O4-C3" }
```

Grupo unirse a

> Une dos secuencias basadas en funciones de selector de teclas y agrupa las coincidencias resultantes para cada elemento.

Sintaxis del método

```
// GroupJoin

var groupJoin = customers.GroupJoin(orders,
                                    c => c.Id,
```

```
                                    o => o.CustomerId,
                                    (c, ors) => c.Name + "-" + string.Join(",", ors.Select(o
=> o.Description)));

// groupJoin = { "C1-O1,O2", "C2-O3", "C3-O4" }
```

Sintaxis de consulta

```
// join … in … on … equals … into …

var groupJoin = from c in customers
                join o in orders
                on c.Id equals o.CustomerId
                into customerOrders
                select string.Join(",", customerOrders.Select(o => o.Description)) + "-" +
c.Name;

// groupJoin = { "O1,O2-C1", "O3-C2", "O4-C3" }
```

Cremallera

Aplica una función específica a los elementos correspondientes de dos secuencias, produciendo una secuencia de los resultados.

```
var numbers = new [] { 1, 2, 3, 4, 5, 6 };
var words = new [] { "one", "two", "three" };

var numbersWithWords =
    numbers
    .Zip(
        words,
        (number, word) => new { number, word });

// Results

//| number | word   |
//| ------ | ------ |
//| 1      |  one   |
//| 2      |  two   |
//| 3      |  three |
```

Operaciones de proyección

La proyección se refiere a las operaciones de transformación de un objeto en una nueva forma.

Seleccionar

Proyecta valores que se basan en una función de transformación.

Sintaxis del método

```
// Select

var numbers = new int[] { 1, 2, 3, 4, 5 };
```

```
var strings = numbers.Select(n => n.ToString());

// strings = { "1", "2", "3", "4", "5" }
```

Sintaxis de consulta

```
// select

var numbers = new int[] { 1, 2, 3, 4, 5 };

var strings = from n in numbers
              select n.ToString();

// strings = { "1", "2", "3", "4", "5" }
```

SelectMany

> Proyecta secuencias de valores que se basan en una función de transformación y luego las aplana en una secuencia.

Sintaxis del método

```
// SelectMany

class Customer
{
    public Order[] Orders { get; set; }
}

class Order
{
    public Order(string desc) { Description = desc; }
    public string Description { get; set; }
}
...

var customers = new Customer[]
{
    new Customer { Orders = new Order[] { new Order("O1"), new Order("O2") } },
    new Customer { Orders = new Order[] { new Order("O3") } },
    new Customer { Orders = new Order[] { new Order("O4") } },
};

var orders = customers.SelectMany(c => c.Orders);

// orders = { Order("O1"), Order("O3"), Order("O3"), Order("O4") }
```

Sintaxis de consulta

```
// multiples from

var orders = from c in customers
             from o in c.Orders
             select o;
```

```
// orders = { Order("O1"), Order("O3"), Order("O3"), Order("O4") }
```

Operaciones de clasificación

Una operación de clasificación ordena los elementos de una secuencia basándose en uno o más atributos.

Orden por

Ordena los valores en orden ascendente.

Sintaxis del método

```
// OrderBy

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = numbers.OrderBy(n => n);

// ordered = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Sintaxis de consulta

```
// orderby

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = from n in numbers
              orderby n
              select n;

// ordered = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

OrderByDescending

Ordena los valores en orden descendente.

Sintaxis del método

```
// OrderByDescending

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = numbers.OrderByDescending(n => n);

// ordered = { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Sintaxis de consulta

```
// orderby

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };
```

```
var ordered = from n in numbers
              orderby n descending
              select n;

// ordered = { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Entonces por

Realiza una clasificación secundaria en orden ascendente.

Sintaxis del método

```
// ThenBy

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = words.OrderBy(w => w.Length).ThenBy(w => w[0]);

// ordered = { "fox", "the", "brown", "jumps", "quick" }
```

Sintaxis de consulta

```
// orderby …, …

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = from w in words
              orderby w.Length, w[0]
              select w;

// ordered = { "fox", "the", "brown", "jumps", "quick" }
```

ThenByDescending

Realiza una clasificación secundaria en orden descendente.

Sintaxis del método

```
// ThenByDescending

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = words.OrderBy(w => w[0]).ThenByDescending(w => w.Length);

// ordered = { "brown", "fox", "jumps", "quick", "the" }
```

Sintaxis de consulta

```
// orderby …, … descending

string[] words = { "the", "quick", "brown", "fox", "jumps" };
```

```
var ordered = from w in words
              orderby w.Length, w[0] descending
              select w;

// ordered = { "the", "fox", "quick", "jumps", "brown" }
```

Marcha atrás

Invierte el orden de los elementos en una colección.

Sintaxis del método

```
// Reverse

var numbers = new int[] { 1, 2, 3, 4, 5 };

var reversed = numbers.Reverse();

// reversed = { 5, 4, 3, 2, 1 }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de conversión

Las operaciones de conversión cambian el tipo de objetos de entrada.

ComoEnumerable

Devuelve la entrada escrita como IEnumerable.

Sintaxis del método

```
// AsEnumerable

int[] numbers = { 1, 2, 3, 4, 5 };

var nums = numbers.AsEnumerable();

// nums: static type is IEnumerable<int>
```

Sintaxis de consulta

```
// Not applicable.
```

AsQueryable

Convierte un IEnumerable en un IQueryable.

Sintaxis del método

```
// AsQueryable

int[] numbers = { 1, 2, 3, 4, 5 };

var nums = numbers.AsQueryable();

// nums: static type is IQueryable<int>
```

Sintaxis de consulta

```
// Not applicable.
```

Emitir

Convierte los elementos de una colección a un tipo especificado.

Sintaxis del método

```
// Cast

var numbers = new object[] { 1, 2, 3, 4, 5 };

var nums = numbers.Cast<int>();

// nums: static type is IEnumerable<int>
```

Sintaxis de consulta

```
// Use an explicitly typed range variable.

var numbers = new object[] { 1, 2, 3, 4, 5 };

var nums = from int n in numbers select n;

// nums: static type is IEnumerable<int>
```

De tipo

Filtra los valores, dependiendo de su capacidad para ser convertidos a un tipo específico.

Sintaxis del método

```
// OfType

var objects = new object[] { 1, "one", 2, "two", 3, "three" };

var numbers = objects.OfType<int>();

// nums = { 1, 2, 3 }
```

Sintaxis de consulta

```
// Not applicable.
```

ToArray

Convierte una colección en una matriz.

Sintaxis del método

```
// ToArray

var numbers = Enumerable.Range(1, 5);

int[] array = numbers.ToArray();

// array = { 1, 2, 3, 4, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Listar

Convierte una colección en una lista.

Sintaxis del método

```
// ToList

var numbers = Enumerable.Range(1, 5);

List<int> list = numbers.ToList();

// list = { 1, 2, 3, 4, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Al diccionario

Pone elementos en un diccionario basado en una función de selector de teclas.

Sintaxis del método

```
// ToDictionary

var numbers = new int[] { 1, 2, 3 };
```

```
var dict = numbers.ToDictionary(n => n.ToString());

// dict = { "1" => 1, "2" => 2, "3" => 3 }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de agregación

Las operaciones de agregación calculan un único valor de una colección de valores.

Agregar

Realiza una operación de agregación personalizada en los valores de una colección.

Sintaxis del método

```
// Aggregate

var numbers = new int[] { 1, 2, 3, 4, 5 };

var product = numbers.Aggregate(1, (acc, n) => acc * n);

// product = 120
```

Sintaxis de consulta

```
// Not applicable.
```

Promedio

Calcula el valor promedio de una colección de valores.

Sintaxis del método

```
// Average

var numbers = new int[] { 1, 2, 3, 4, 5 };

var average = numbers.Average();

// average = 3
```

Sintaxis de consulta

```
// Not applicable.
```

Contar

Cuenta los elementos en una colección, opcionalmente solo aquellos elementos que satisfacen una función de predicado.

Sintaxis del método

```
// Count

var numbers = new int[] { 1, 2, 3, 4, 5 };

int count = numbers.Count(n => n % 2 == 0);

// count = 2
```

Sintaxis de consulta

```
// Not applicable.
```

LongCount

Cuenta los elementos en una colección grande, opcionalmente solo aquellos elementos que satisfacen una función de predicado.

Sintaxis del método

```
// LongCount

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

long count = numbers.LongCount();

// count = 10
```

Sintaxis de consulta

```
// Not applicable.
```

Max

Determina el valor máximo en una colección. Lanza una excepción si la colección está vacía.

Sintaxis del método

```
// Max

var numbers = new int[] { 1, 2, 3, 4, 5 };

var max = numbers.Max();

// max = 5
```

Sintaxis de consulta

```
// Not applicable.
```

Min

Determina el valor mínimo en una colección. Lanza una excepción si la colección está vacía.

Sintaxis del método

```
// Min

var numbers = new int[] { 1, 2, 3, 4, 5 };

var min = numbers.Min();

// min = 1
```

Sintaxis de consulta

```
// Not applicable.
```

Min- / MaxOrDefault

A diferencia de otras extensiones LinQ, `Min()` y `Max()` no tienen una sobrecarga sin excepciones. Por lo tanto, el `IEnumerable` debe verificarse en `Any()` antes de llamar a `Min()` o `Max()`

```
// Max

var numbers = new int[] { };

var max = numbers.Any() ? numbers.Max() : 0;

// max = 0
```

Suma

Calcula la suma de los valores en una colección.

Sintaxis del método

```
// Sum

var numbers = new int[] { 1, 2, 3, 4, 5 };

var sum = numbers.Sum();

// sum = 15
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de cuantificación

Las operaciones de cuantificación devuelven un valor booleano que indica si algunos o todos los elementos en una secuencia satisfacen una condición.

Todos

Determina si todos los elementos en una secuencia satisfacen una condición.

Sintaxis del método

```
// All

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool areLessThan10 = numbers.All(n => n < 10);

// areLessThan10 = true
```

Sintaxis de consulta

```
// Not applicable.
```

Alguna

Determina si algún elemento en una secuencia satisface una condición.

Sintaxis del método

```
// Any

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool anyOneIsEven = numbers.Any(n => n % 2 == 0);

// anyOneIsEven = true
```

Sintaxis de consulta

```
// Not applicable.
```

Contiene

Determina si una secuencia contiene un elemento especificado.

Sintaxis del método

```
// Contains

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool appears = numbers.Contains(10);

// appears = false
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de agrupación

La agrupación se refiere a las operaciones de poner datos en grupos para que los elementos de cada grupo compartan un atributo común.

Agrupar por

Agrupa elementos que comparten un atributo común.

Sintaxis del método

```
// GroupBy

class Order
{
    public string Customer { get; set; }
    public string Description { get; set; }
}
...

var orders = new Order[]
{
    new Order { Customer = "C1", Description = "O1" },
    new Order { Customer = "C2", Description = "O2" },
    new Order { Customer = "C3", Description = "O3" },
    new Order { Customer = "C1", Description = "O4" },
    new Order { Customer = "C1", Description = "O5" },
    new Order { Customer = "C3", Description = "O6" },
};

var groups = orders.GroupBy(o => o.Customer);

// groups: { (Key="C1", Values="O1","O4","O5"), (Key="C2", Values="O2"), (Key="C3",
Values="O3","O6") }
```

Sintaxis de consulta

```
// group … by

var groups = from o in orders
```

```
            group o by o.Customer;

// groups: { (Key="C1", Values="O1","O4","O5"), (Key="C2", Values="O2"), (Key="C3",
Values="O3","O6") }
```

Para buscar

> Inserta elementos en un diccionario de uno a varios basado en una función de selector de teclas.

Sintaxis del método

```
// ToLookUp

var ordersByCustomer = orders.ToLookup(o => o.Customer);

// ordersByCustomer = ILookUp<string, Order>
// {
//     "C1" => { Order("01"), Order("04"), Order("05") },
//     "C2" => { Order("02") },
//     "C3" => { Order("03"), Order("06") }
// }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de partición

La partición se refiere a las operaciones de dividir una secuencia de entrada en dos secciones, sin reorganizar los elementos, y luego devolver una de las secciones.

Omitir

> Salta elementos hasta una posición específica en una secuencia.

Sintaxis del método

```
// Skip

var numbers = new int[] { 1, 2, 3, 4, 5 };

var skipped = numbers.Skip(3);

// skipped = { 4, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

SkipWhile

> Omite elementos basados en una función de predicado hasta que un elemento no cumpla la condición.

Sintaxis del método

```
// Skip

var numbers = new int[] { 1, 3, 5, 2, 1, 3, 5 };

var skipLeadingOdds = numbers.SkipWhile(n => n % 2 != 0);

// skipLeadingOdds = { 2, 1, 3, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Tomar

> Lleva los elementos hasta una posición específica en una secuencia.

Sintaxis del método

```
// Take

var numbers = new int[] { 1, 2, 3, 4, 5 };

var taken = numbers.Take(3);

// taken = { 1, 2, 3 }
```

Sintaxis de consulta

```
// Not applicable.
```

TakeWhile

> Toma elementos basados en una función de predicado hasta que un elemento no satisface la condición.

Sintaxis del método

```
// TakeWhile

var numbers = new int[] { 1, 3, 5, 2, 1, 3, 5 };

var takeLeadingOdds = numbers.TakeWhile(n => n % 2 != 0);

// takeLeadingOdds = { 1, 3, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de generacion

Generación se refiere a la creación de una nueva secuencia de valores.

DefaultIfEmpty

Reemplaza una colección vacía con una colección singleton de valor predeterminado.

Sintaxis del método

```
// DefaultIfEmpty

var nums = new int[0];

var numbers = nums.DefaultIfEmpty();

// numbers = { 0 }
```

Sintaxis de consulta

```
// Not applicable.
```

Vacío

Devuelve una colección vacía.

Sintaxis del método

```
// Empty

var empty = Enumerable.Empty<string>();

// empty = IEnumerable<string> { }
```

Sintaxis de consulta

```
// Not applicable.
```

Distancia

Genera una colección que contiene una secuencia de números.

Sintaxis del método

```
// Range
```

```
var range = Enumerable.Range(1, 5);

// range = { 1, 2, 3, 4, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Repetir

Genera una colección que contiene un valor repetido.

Sintaxis del método

```
// Repeat

var repeats = Enumerable.Repeat("s", 3);

// repeats = { "s", "s", "s" }
```

Sintaxis de consulta

```
// Not applicable.
```

Establecer Operaciones

Las operaciones de conjunto se refieren a las operaciones de consulta que producen un conjunto de resultados que se basa en la presencia o ausencia de elementos equivalentes dentro de las mismas colecciones o conjuntos (o conjuntos).

Distinto

Elimina valores duplicados de una colección.

Sintaxis del método

```
// Distinct

var numbers = new int[] { 1, 2, 3, 1, 2, 3 };

var distinct = numbers.Distinct();

// distinct = { 1, 2, 3 }
```

Sintaxis de consulta

```
// Not applicable.
```

Excepto

Devuelve la diferencia establecida, lo que significa que los elementos de una colección no aparecen en una segunda colección.

Sintaxis del método

```
// Except

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };

var except = numbers1.Except(numbers2);

// except = { 1, 2, 3 }
```

Sintaxis de consulta

```
// Not applicable.
```

Intersecarse

Devuelve la intersección del conjunto, lo que significa que los elementos que aparecen en cada una de las dos colecciones.

Sintaxis del método

```
// Intersect

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };

var intersect = numbers1.Intersect(numbers2);

// intersect = { 4, 5 }
```

Sintaxis de consulta

```
// Not applicable.
```

Unión

Devuelve la unión establecida, lo que significa elementos únicos que aparecen en cualquiera de las dos colecciones.

Sintaxis del método

```
// Union

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };
```

```
var union = numbers1.Union(numbers2);

// union = { 1, 2, 3, 4, 5, 6, 7, 8 }
```

Sintaxis de consulta

```
// Not applicable.
```

Operaciones de igualdad

Dos secuencias cuyos elementos correspondientes son iguales y que tienen el mismo número de elementos se consideran iguales.

SecuenciaEqual

> Determina si dos secuencias son iguales comparando elementos de una manera por pares.

Sintaxis del método

```
// SequenceEqual

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 1, 2, 3, 4, 5 };

var equals = numbers1.SequenceEqual(numbers2);

// equals = true
```

Sintaxis de consulta

```
// Not Applicable.
```

Operaciones de elementos

Las operaciones de elementos devuelven un único elemento específico de una secuencia.

Elemento

> Devuelve el elemento en un índice especificado en una colección.

Sintaxis del método

```
// ElementAt

var strings = new string[] { "zero", "one", "two", "three" };

var str = strings.ElementAt(2);

// str = "two"
```

Sintaxis de consulta

```
// Not Applicable.
```

ElementAtOrDefault

> Devuelve el elemento en un índice especificado en una colección o un valor predeterminado si el índice está fuera de rango.

Sintaxis del método

```
// ElementAtOrDefault

var strings = new string[] { "zero", "one", "two", "three" };

var str = strings.ElementAtOrDefault(10);

// str = null
```

Sintaxis de consulta

```
// Not Applicable.
```

---

primero

> Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición.

Sintaxis del método

```
// First

var numbers = new int[] { 1, 2, 3, 4, 5 };

var first = strings.First();

// first = 1
```

Sintaxis de consulta

```
// Not Applicable.
```

FirstOrDefault

> Devuelve el primer elemento de una colección, o el primer elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento.

Sintaxis del método

```
// FirstOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var firstGreaterThanTen = strings.FirstOrDefault(n => n > 10);

// firstGreaterThanTen = 0
```

Sintaxis de consulta

```
// Not Applicable.
```

Último

Devuelve el último elemento de una colección, o el último elemento que satisface una condición.

Sintaxis del método

```
// Last

var numbers = new int[] { 1, 2, 3, 4, 5 };

var last = strings.Last();

// last = 5
```

Sintaxis de consulta

```
// Not Applicable.
```

LastOrDefault

Devuelve el último elemento de una colección, o el último elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento.

Sintaxis del método

```
// LastOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var lastGreaterThanTen = strings.LastOrDefault(n => n > 10);

// lastGreaterThanTen = 0
```

Sintaxis de consulta

```
// Not Applicable.
```

Soltero

> Devuelve el único elemento de una colección, o el único elemento que satisface una condición.

Sintaxis del método

```
// Single

var numbers = new int[] { 1 };

var single = strings.Single();

// single = 1
```

Sintaxis de consulta

```
// Not Applicable.
```

SingleOrDefault

> Devuelve el único elemento de una colección, o el único elemento que satisface una condición. Devuelve un valor predeterminado si no existe tal elemento o la colección no contiene exactamente un elemento.

Sintaxis del método

```
// SingleOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var singleGreaterThanFour = strings.SingleOrDefault(n => n > 4);

// singleGreaterThanFour = 5
```

Sintaxis de consulta

```
// Not Applicable.
```

Lea Operadores de consultas estándar en línea:
https://riptutorial.com/es/linq/topic/2535/operadores-de-consultas-estandar

Creditos

S. No	Capítulos	Contributors
1	Empezando con linq	AlexFoxGill, Arturo Menchaca, Colin Young, Community, David B, flindeberg, Ivan Yurchenko, Joshua Poling, Kobi, Mark Hurd, Matthew Haugen, meJustAndrew, mmushtaq, Peter Mortensen, Richard Everett, Ryan Abbott, Tom Wuyts, Travis J, Wyck, Zev Spitz
2	Linq usando Take while y Skip mientras que	kari kalan
3	Modos de ejecución de métodos: transmisión inmediata, diferida, sin transmisión diferida	Colin Young, mmushtaq, Travis J, Zev Spitz
4	Operadores de consultas estándar	Arturo Menchaca, James Cockayne, Mark Hurd, Toxantron