EBook Gratuito

APPRENDIMENTO

linq

Free unaffiliated eBook created from
Stack Overflow contributors.

#linq

Sommario

Di

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

Capitolo 1: Iniziare con linq

Osservazioni

LINQ è un insieme di funzionalità introdotte in .NET Framework versione 3.5 che colma il divario tra il mondo degli oggetti e il mondo dei dati.

Tradizionalmente, le query sui dati sono espresse come semplici stringhe senza controllo del tipo in fase di compilazione o supporto IntelliSense. Inoltre, è necessario apprendere un linguaggio di query diverso per ciascun tipo di origine dati: database SQL, documenti XML, vari servizi Web e così via. LINQ crea una query in un costrutto linguistico di prima classe in C # e Visual Basic. Si scrivono query contro raccolte di oggetti fortemente tipizzate utilizzando parole chiave della lingua e operatori familiari.

Examples

Impostare

LINQ richiede .NET 3.5 o versione successiva (oppure .NET 2.0 con LINQBridge).

Aggiungi un riferimento a **System.Core** , se non è stato ancora aggiunto.

Nella parte superiore del file, importa lo spazio dei nomi:

- C #

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

- VB.NET

```
Imports System.Linq
```

I diversi join in LINQ

Nei seguenti esempi, utilizzeremo i seguenti esempi:

```
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,
  },
};
```

INNER JOIN

Sintassi delle query

```
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();
```

Sintassi del metodo

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

Risultato:

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

SINISTRA ESTERNO

```
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();
```

Risultato:

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

CROSS JOIN

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

Risultato:

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

GROUP JOIN

```
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();
```

Gli `Orders` Propertie ora contengono un oggetto `IEnumerable<Order>` con tutti gli ordini collegati.

Risultato:

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

Come aderire a più condizioni

Quando ti unisci a una singola condizione, puoi usare:

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

Quando ti unisci a più persone, usa:

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

Assicurati che entrambi gli oggetti anonimi abbiano le stesse proprietà, e in VB.NET, devono essere contrassegnati con `Key` , anche se VB.NET consente più clausole `Equals` separate da `And` :

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

Sintassi delle query e sintassi del metodo

La sintassi delle query e la sintassi del metodo sono semanticamente identiche, ma molte persone trovano la sintassi delle query più semplice e più facile da leggere. Diciamo che dobbiamo recuperare tutti gli oggetti pari ordinati in ordine crescente da una serie di numeri.

C #:

```
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)
```

Ricorda che alcune query **devono** essere espresse come chiamate di metodo. Ad esempio, è necessario utilizzare una chiamata al metodo per esprimere una query che recupera il numero di elementi che corrispondono a una condizione specificata. È inoltre necessario utilizzare una chiamata al metodo per una query che recupera l'elemento che ha il valore massimo in una sequenza di origine. Quindi questo potrebbe essere un vantaggio dell'uso della sintassi del metodo per rendere il codice più coerente. Tuttavia, ovviamente è sempre possibile applicare il metodo dopo una chiamata di sintassi di query:

C #:

```
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();
```

Metodi LINQ e IEnumerable vs IQueryable

I metodi di estensione LINQ su `IEnumerable<T>` prendono i metodi attuali [1] , sia che si tratti di metodi anonimi:

```
//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 metodi con nome (metodi definiti esplicitamente come parte di una classe):

```
//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
```

In teoria, è possibile analizzare l'IL del metodo , capire che cosa il metodo sta tentando di fare e applicare la logica di quel metodo a qualsiasi origine dati sottostante, non solo oggetti in memoria. Ma analizzare IL non è per i deboli di cuore.

Fortunatamente, .NET fornisce l' `IQueryable<T>` e i metodi di estensione a `System.Linq.Queryable` , per questo scenario. Questi metodi di estensione prendono un albero di espressioni - una struttura dati che rappresenta il codice - invece di un metodo effettivo, che il provider LINQ può quindi analizzare [2] e convertire in un modulo più appropriato per interrogare l'origine dati sottostante. Per esempio:

```
//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)
```

Se (ad esempio) questa query è su un database SQL, il provider potrebbe convertire questa espressione nella seguente istruzione SQL:

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

ed eseguirlo contro la fonte dei dati.

D'altra parte, se la query è contro un'API REST, il provider può convertire la stessa espressione in una chiamata API:

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

Ci sono due principali vantaggi nel personalizzare una richiesta di dati basata su un'espressione (anziché caricare l'intera raccolta in memoria e interrogare localmente):

- L'origine dati sottostante può spesso eseguire query in modo più efficiente. Ad esempio, potrebbe esserci un indice su `LastName` . Caricare gli oggetti nella memoria locale e interrogare in memoria perde quell'efficienza.
- I dati possono essere modellati e ridotti prima di essere trasferiti. In questo caso, il database / servizio web deve solo restituire i dati corrispondenti, in contrapposizione all'intero insieme di Persone disponibili dall'origine dati.

Gli appunti

1. Tecnicamente, in realtà non prendono metodi, ma delegano le istanze che puntano a metodi . Tuttavia, questa distinzione è irrilevante qui.
2. Questo è il motivo per cui errori come " *LINQ to Entities" non riconosce il metodo "System.String ToString ()" e questo metodo non può essere tradotto in un'espressione di archivio.* ". Il provider LINQ (in questo caso il provider Entity Framework) non sa come analizzare e tradurre una chiamata a `ToString` in SQL equivalente.

Leggi Iniziare con linq online: https://riptutorial.com/it/linq/topic/842/iniziare-con-linq

Capitolo 2: Linq Uso Take take e Skip While

introduzione

Take, Skip, TakeWhile e SkipWhile sono tutti chiamati operatori di partizionamento poiché ottengono una sezione di una sequenza di input determinata da una determinata condizione. Cerchiamo di discutere di questi operatori

Examples

Prendi il metodo

Il metodo Take Porta gli elementi fino a una posizione specificata a partire dal primo elemento di una sequenza. **Firma di Take:**

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

Esempio:

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

Produzione:

Il risultato è 1,5,8,4 e 9 per ottenere cinque elementi.

Salta Metodo

Salta gli elementi fino a una posizione specificata a partire dal primo elemento di una sequenza.

Firma di Skip:

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

Esempio

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

Uscita: il risultato è 3,6,7,2 e 0 per ottenere l'elemento.

TakeWhile ():

Restituisce gli elementi dalla raccolta specificata fino a quando la condizione specificata è vera. Se il primo elemento non soddisfa la condizione, restituisce una raccolta vuota.

Firma di TakeWhile ():

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

Un'altra firma di carico eccessivo:

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

Esempio I:

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

Produzione:

Restituirà Di eleament 1,5,8 e 4

Esempio II:

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

Produzione:

Restituirà degli elementi 1,2,3 e 4

SkipWhile ()

Salta gli elementi in base a una condizione finché un elemento non soddisfa la condizione. Se il primo elemento non soddisfa la condizione, salta quindi 0 elementi e restituisce tutti gli elementi nella sequenza.

Firma di SkipWhile ():

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

Un'altra firma di carico eccessivo:

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

Esempio I:

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

Produzione:

Restituirà degli elementi 9,3,6,7,2 e 0.

Esempio II:

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

Produzione:

Restituirà degli elementi 1,9,3,6,7,2 e 0.

Leggi Linq Uso Take take e Skip While online: https://riptutorial.com/it/linq/topic/10810/linq-uso-take-take-e-skip-while

Capitolo 3: Modalità di esecuzione del metodo: streaming immediato, differito, differito non streaming

Examples

Esecuzione differita vs esecuzione immediata

Alcuni metodi LINQ restituiscono un oggetto query. Questo oggetto non contiene i risultati della query; invece, ha tutte le informazioni necessarie per generare quei risultati:

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

La query contiene una chiamata a `Console.Write` , ma non è stato prodotto alcun output sulla console. Questo perché la query non è stata ancora eseguita, e quindi la funzione passata a `Select` non è mai stata valutata. Questa operazione è nota come **esecuzione differita: l'esecuzione** della query viene posticipata fino a un momento successivo.

Altri metodi LINQ forzano **un'esecuzione immediata** della query; questi metodi eseguono la query e ne generano i valori:

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

A questo punto, la funzione passata in `Select` verrà valutata per ogni valore nell'elenco originale, e quanto segue verrà emesso sulla console:

    1 2 3 4 5

In genere, i metodi LINQ che restituiscono un singolo valore (come `Max` o `Count`) o che restituiscono un oggetto che contiene effettivamente i valori (come `ToList` o `ToDictionary`) vengono eseguiti immediatamente.

I metodi che restituiscono un oggetto `IEnumerable<T>` o `IQueryable<T>` restituiscono l'oggetto query e consentono di posticipare l'esecuzione fino a un punto successivo.

Se un particolare metodo LINQ impone una query da eseguire immediatamente o meno, è possibile trovarlo su MSDN - C # o VB.NET .

Modalità streaming (valutazione lazy) vs modalità non streaming (valutazione

stimolante)

Tra i metodi LINQ che utilizzano l'esecuzione differita, alcuni richiedono un singolo valore da valutare alla volta. Il seguente codice:

```
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}");
}
```

produrrà:

3
foreach valore di iterazione: 3
5
foreach valore di iterazione: 5
1
foreach valore di iterazione: 1
2
foreach valore di iterazione: 2

poiché la funzione passata a `Select` viene valutata ad ogni iterazione di `foreach` . Questo è noto come **modalità di streaming** o **valutazione lazy** .

Altri metodi LINQ - operatori di ordinamento e raggruppamento - richiedono che *tutti* i valori siano valutati, prima che possano restituire *qualsiasi* valore:

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

produrrà:

3
5
1
2
foreach valore di iterazione: 1
foreach valore di iterazione: 2
foreach valore di iterazione: 3
foreach valore di iterazione: 5

In questo caso, poiché i valori devono essere generati nel `foreach` in ordine crescente, tutti gli elementi devono prima essere valutati, per determinare quale è il più piccolo, e che è il successivo più piccolo, e così via. Questo è noto come **modalità non streaming** o **valutazione entusiasta** .

Se un particolare metodo LINQ utilizza la modalità streaming o non streaming, è possibile trovarlo su MSDN - C # o VB.NET .

Vantaggi dell'esecuzione posticipata: creazione di query

L'esecuzione posticipata consente di combinare diverse operazioni per creare la query finale, prima di valutare i valori:

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

Se eseguiamo la query a questo punto:

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

otterremmo il seguente risultato:

    2 2 3 4 6 9

Ma possiamo modificare la query aggiungendo più operatori:

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

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

Produzione:

    20 20 40 60

Vantaggi dell'esecuzione posticipata - interrogazione dei dati correnti

Con l'esecuzione posticipata, se i dati da interrogare vengono modificati, l'oggetto query utilizza i dati al momento dell'esecuzione, non al momento della definizione.

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

Se eseguiamo la query a questo punto con un metodo immediato o `foreach` , la query opererà nell'elenco dei numeri pari.

Tuttavia, se cambiamo i valori nell'elenco:

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

o anche se assegniamo una nuova lista ai `data` :

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

e quindi eseguire la query, la query opererà sul nuovo valore dei `data` :

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

e mostrerà quanto segue:

    1 9 25 49 81

Leggi Modalità di esecuzione del metodo: streaming immediato, differito, differito non streaming online: https://riptutorial.com/it/linq/topic/7102/modalita-di-esecuzione-del-metodo--streaming-immediato--differito--differito-non-streaming

Capitolo 4: Operatori di query standard

Osservazioni

Le query Linq vengono scritte utilizzando gli Operatori di query standard (che sono un insieme di metodi di estensione che operano principalmente su oggetti di tipo `IEnumerable<T>` e `IQueryable<T>`) o utilizzando le espressioni di query (che in fase di compilazione vengono convertite in Operatore di query standard chiamate di metodo).

Gli operatori di query forniscono funzionalità di query tra cui filtraggio, proiezione, aggregazione, ordinamento e altro ancora.

Examples

Operazioni di concatenazione

La concatenazione si riferisce all'operazione di accodare una sequenza a un'altra.

concat

> Concatena due sequenze per formare una sequenza.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// Not applicable.
```

Operazioni di filtraggio

Il filtraggio si riferisce alle operazioni di restrizione del set di risultati per contenere solo quegli elementi che soddisfano una condizione specificata.

Dove

> Seleziona i valori che sono basati su una funzione di predicato.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// 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 }
```

OfType

> Seleziona i valori, a seconda della loro capacità di essere espressi su un tipo specificato.

Sintassi del metodo

```
// OfType

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

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

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

Sintassi delle query

```
// Not applicable.
```

Partecipa alle operazioni

Un join di due origini dati è l'associazione di oggetti in un'origine dati con oggetti che condividono un attributo comune in un'altra origine dati.

Aderire

> Unisce due sequenze basate sulle funzioni di selezione chiave ed estrae coppie di valori.

Sintassi del metodo

```
// 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" }
```

Sintassi delle query

```
// 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" }
```

GroupJoin

> Unisce due sequenze basate sulle funzioni di selezione chiave e raggruppa le corrispondenze risultanti per ciascun elemento.

Sintassi del metodo

```
// 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" }
```

Sintassi delle query

```
// 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" }
```

Cerniera lampo

> Applica una funzione specificata agli elementi corrispondenti di due sequenze, producendo una sequenza dei risultati.

```
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 |
```

Operazioni di proiezione

La proiezione si riferisce alle operazioni di trasformazione di un oggetto in una nuova forma.

Selezionare

> Valori dei progetti basati su una funzione di trasformazione.

Sintassi del metodo

```
// Select

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

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

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

Sintassi delle query

```
// 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

> Progetta sequenze di valori basati su una funzione di trasformazione e quindi li appiattisce in un'unica sequenza.

Sintassi del metodo

```
// 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") }
```

Sintassi delle query

```
// multiples from

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

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

Operazioni di ordinamento

Un'operazione di ordinamento ordina gli elementi di una sequenza in base a uno o più attributi.

Ordinato da

Ordina i valori in ordine crescente.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// 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

Ordina i valori in ordine decrescente.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// 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 }
```

ThenBy

Esegue un ordinamento secondario in ordine crescente.

Sintassi del metodo

```
// 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" }
```

Sintassi delle query

```
// 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

Esegue un ordinamento secondario in ordine decrescente.

Sintassi del metodo

```
// 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" }
```

Sintassi delle query

```
// 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" }
```

Inverso

Inverte l'ordine degli elementi in una raccolta.

Sintassi del metodo

```
// Reverse

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

var reversed = numbers.Reverse();

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

Sintassi delle query

```
// Not applicable.
```

Operazioni di conversione

Le operazioni di conversione cambiano il tipo di oggetti di input.

AsEnumerable

Restituisce l'input digitato come IEnumerable.

Sintassi del metodo

```
// AsEnumerable

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

var nums = numbers.AsEnumerable();

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

Sintassi delle query

```
// Not applicable.
```

AsQueryable

Converte un oggetto IEnumerable in IQueryable.

Sintassi del metodo

```
// AsQueryable
```

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

var nums = numbers.AsQueryable();

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

Sintassi delle query

```
// Not applicable.
```

---

lanciare

Casta gli elementi di una raccolta in un tipo specificato.

Sintassi del metodo

```
// Cast

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

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

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

Sintassi delle query

```
// 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>
```

OfType

Valori dei filtri, in base alla loro capacità di essere espressi su un tipo specificato.

Sintassi del metodo

```
// OfType

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

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

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

Sintassi delle query

```
// Not applicable.
```

ToArray

Converte una raccolta in una matrice.

Sintassi del metodo

```
// ToArray

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

int[] array = numbers.ToArray();

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

Sintassi delle query

```
// Not applicable.
```

Elencare

Converte una collezione in una lista.

Sintassi del metodo

```
// ToList

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

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

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

Sintassi delle query

```
// Not applicable.
```

ToDictionary

Inserisce elementi in un dizionario basato su una funzione di selezione chiave.

Sintassi del metodo

```
// ToDictionary

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

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

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

Sintassi delle query

```
// Not applicable.
```

Operazioni di aggregazione

Le operazioni di aggregazione calcolano un singolo valore da un insieme di valori.

Aggregato

Esegue un'operazione di aggregazione personalizzata sui valori di una raccolta.

Sintassi del metodo

```
// Aggregate

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

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

// product = 120
```

Sintassi delle query

```
// Not applicable.
```

Media

Calcola il valore medio di un insieme di valori.

Sintassi del metodo

```
// Average

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

var average = numbers.Average();

// average = 3
```

Sintassi delle query

```
// Not applicable.
```

Contare

Conta gli elementi in una raccolta, opzionalmente solo quegli elementi che soddisfano una funzione di predicato.

Sintassi del metodo

```
// Count

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

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

// count = 2
```

Sintassi delle query

```
// Not applicable.
```

LongCount

> Conta gli elementi in una grande raccolta, opzionalmente solo quegli elementi che soddisfano una funzione di predicato.

Sintassi del metodo

```
// LongCount

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

long count = numbers.LongCount();

// count = 10
```

Sintassi delle query

```
// Not applicable.
```

Max

> Determina il valore massimo in una raccolta. Genera un'eccezione se la raccolta è vuota.

Sintassi del metodo

```
// Max

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

var max = numbers.Max();

// max = 5
```

Sintassi delle query

```
// Not applicable.
```

---

min

> Determina il valore minimo in una raccolta. Genera un'eccezione se la raccolta è vuota.

Sintassi del metodo

```
// Min

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

var min = numbers.Min();

// min = 1
```

Sintassi delle query

```
// Not applicable.
```

Min- / MaxOrDefault

> A differenza di altre estensioni LinQ `Min()` e `Max()` non hanno un sovraccarico senza eccezioni. Pertanto, è necessario selezionare `IEnumerable Any()` prima di chiamare `Min()` o `Max()`

```
// Max

var numbers = new int[] { };

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

// max = 0
```

Somma

> Calcola la somma dei valori in una raccolta.

Sintassi del metodo

```
// Sum

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

var sum = numbers.Sum();

// sum = 15
```

Sintassi delle query

```
// Not applicable.
```

Operazioni di Quantificatore

Le operazioni Quantifier restituiscono un valore booleano che indica se alcuni o tutti gli elementi di una sequenza soddisfano una condizione.

Tutti

Determina se tutti gli elementi di una sequenza soddisfano una condizione.

Sintassi del metodo

```
// All

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

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

// areLessThan10 = true
```

Sintassi delle query

```
// Not applicable.
```

Qualunque

Determina se alcuni elementi in una sequenza soddisfano una condizione.

Sintassi del metodo

```
// Any

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

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

// anyOneIsEven = true
```

Sintassi delle query

```
// Not applicable.
```

---

contiene

Determina se una sequenza contiene un elemento specificato.

Sintassi del metodo

```
// Contains

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

```
bool appears = numbers.Contains(10);

// appears = false
```

Sintassi delle query

```
// Not applicable.
```

Operazioni di raggruppamento

Raggruppamento si riferisce alle operazioni di inserimento dei dati in gruppi in modo che gli elementi di ciascun gruppo condividano un attributo comune.

Raggruppa per

Gruppi di elementi che condividono un attributo comune.

Sintassi del metodo

```
// 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") }
```

Sintassi delle query

```
// 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") }
```

ToLookup

Inserisce elementi in un dizionario uno-a-molti basato su una funzione di selezione chiave.

Sintassi del metodo

```
// 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") }
// }
```

Sintassi delle query

```
// Not applicable.
```

Operazioni di partizione

Il partizionamento si riferisce alle operazioni di divisione di una sequenza di input in due sezioni, senza riorganizzare gli elementi e quindi restituire una delle sezioni.

Salta

Salta gli elementi fino a una posizione specificata in una sequenza.

Sintassi del metodo

```
// Skip

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

var skipped = numbers.Skip(3);

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

Sintassi delle query

```
// Not applicable.
```

SkipWhile

Salta gli elementi in base a una funzione predicato fino a quando un elemento non soddisfa la condizione.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// Not applicable.
```

Prendere

> Porta gli elementi fino a una posizione specificata in una sequenza.

Sintassi del metodo

```
// Take

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

var taken = numbers.Take(3);

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

Sintassi delle query

```
// Not applicable.
```

TakeWhile

> Accetta elementi in base a una funzione di predicato fino a quando un elemento non soddisfa la condizione.

Sintassi del metodo

```
// TakeWhile

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

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

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

Sintassi delle query

```
// Not applicable.
```

Operazioni generazionali

La generazione si riferisce alla creazione di una nuova sequenza di valori.

DefaultIfEmpty

Sostituisce una raccolta vuota con una raccolta singleton con valore predefinito.

Sintassi del metodo

```
// DefaultIfEmpty

var nums = new int[0];

var numbers = nums.DefaultIfEmpty();

// numbers = { 0 }
```

Sintassi delle query

```
// Not applicable.
```

Vuoto

Restituisce una collezione vuota.

Sintassi del metodo

```
// Empty

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

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

Sintassi delle query

```
// Not applicable.
```

Gamma

Genera una collezione che contiene una sequenza di numeri.

Sintassi del metodo

```
// Range

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

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

Sintassi delle query

```
// Not applicable.
```

Ripetere

Genera una collezione che contiene un valore ripetuto.

Sintassi del metodo

```
// Repeat

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

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

Sintassi delle query

```
// Not applicable.
```

Imposta le operazioni

Le operazioni di set si riferiscono a operazioni di query che producono un set di risultati basato sulla presenza o l'assenza di elementi equivalenti all'interno delle raccolte o insiemi uguali o separati.

distinto

Rimuove i valori duplicati da una raccolta.

Sintassi del metodo

```
// Distinct

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

var distinct = numbers.Distinct();

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

Sintassi delle query

```
// Not applicable.
```

---

tranne

Restituisce la differenza impostata, ovvero gli elementi di una raccolta che non appaiono in una seconda raccolta.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// Not applicable.
```

---

intersecare

> Restituisce l'intersezione impostata, ovvero gli elementi che appaiono in ciascuna delle due raccolte.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// Not applicable.
```

Unione

> Restituisce l'unione dell'insieme, che significa elementi unici che appaiono in una delle due raccolte.

Sintassi del metodo

```
// 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 }
```

Sintassi delle query

```
// Not applicable.
```

Operazioni di uguaglianza

Due sequenze i cui elementi corrispondenti sono uguali e che hanno lo stesso numero di elementi sono considerate uguali.

SequenceEqual

> Determina se due sequenze sono uguali confrontando gli elementi in un modo saggio coppia.

Sintassi del metodo

```
// 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
```

Sintassi delle query

```
// Not Applicable.
```

Operazioni sugli elementi

Le operazioni sugli elementi restituiscono un singolo elemento specifico da una sequenza.

ElementAt

> Restituisce l'elemento in un indice specificato in una raccolta.

Sintassi del metodo

```
// ElementAt

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

var str = strings.ElementAt(2);

// str = "two"
```

Sintassi delle query

```
// Not Applicable.
```

ElementAtOrDefault

Restituisce l'elemento in un indice specificato in una raccolta o un valore predefinito se l'indice non è compreso nell'intervallo.

Sintassi del metodo

```
// ElementAtOrDefault

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

var str = strings.ElementAtOrDefault(10);

// str = null
```

Sintassi delle query

```
// Not Applicable.
```

Primo

Restituisce il primo elemento di una raccolta o il primo elemento che soddisfa una condizione.

Sintassi del metodo

```
// First

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

var first = strings.First();

// first = 1
```

Sintassi delle query

```
// Not Applicable.
```

FirstOrDefault

Restituisce il primo elemento di una raccolta o il primo elemento che soddisfa una condizione. Restituisce un valore predefinito se non esiste alcun elemento di questo tipo.

Sintassi del metodo

```
// FirstOrDefault

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

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

// firstGreaterThanTen = 0
```

Sintassi delle query

```
// Not Applicable.
```

Scorso

Restituisce l'ultimo elemento di una raccolta o l'ultimo elemento che soddisfa una condizione.

Sintassi del metodo

```
// Last

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

var last = strings.Last();

// last = 5
```

Sintassi delle query

```
// Not Applicable.
```

LastOrDefault

Restituisce l'ultimo elemento di una raccolta o l'ultimo elemento che soddisfa una condizione. Restituisce un valore predefinito se non esiste alcun elemento di questo tipo.

Sintassi del metodo

```
// LastOrDefault

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

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

// lastGreaterThanTen = 0
```

Sintassi delle query

```
// Not Applicable.
```

---

singolo

Restituisce l'unico elemento di una raccolta o l'unico elemento che soddisfa una condizione.

Sintassi del metodo

```
// Single

var numbers = new int[] { 1 };

var single = strings.Single();

// single = 1
```

Sintassi delle query

```
// Not Applicable.
```

SingleOrDefault

> Restituisce l'unico elemento di una raccolta o l'unico elemento che soddisfa una condizione. Restituisce un valore predefinito se non esiste alcun elemento di questo tipo o la raccolta non contiene esattamente un elemento.

Sintassi del metodo

```
// SingleOrDefault

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

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

// singleGreaterThanFour = 5
```

Sintassi delle query

```
// Not Applicable.
```

Leggi Operatori di query standard online: https://riptutorial.com/it/linq/topic/2535/operatori-di-query-standard

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare 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 Uso Take take e Skip While	kari kalan
3	Modalità di esecuzione del metodo: streaming immediato, differito, differito non streaming	Colin Young, mmushtaq, Travis J, Zev Spitz
4	Operatori di query standard	Arturo Menchaca, James Cockayne, Mark Hurd, Toxantron