eBook Gratuit

APPRENEZ linq

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#linq

Table des matières

À propos

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

Chapitre 1: Démarrer avec linq

Remarques

LINQ est un ensemble de fonctionnalités introduites dans le .NET Framework version 3.5 qui comble le fossé entre le monde des objets et le monde des données.

Traditionnellement, les requêtes sur les données sont exprimées sous forme de chaînes simples sans vérification de type au moment de la compilation ou du support IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données: bases de données SQL, documents XML, divers services Web, etc. LINQ fait une requête une construction de langage de première classe en C # et Visual Basic. Vous écrivez des requêtes sur des collections d'objets fortement typées en utilisant des mots-clés de langage et des opérateurs familiers.

Examples

Installer

LINQ nécessite .NET 3.5 ou version ultérieure (ou .NET 2.0 avec LINQBridge).

Ajoutez une référence à **System.Core** , si elle n'a pas encore été ajoutée.

En haut du fichier, importez l'espace de noms:

- C #

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

- VB.NET

```
Imports System.Linq
```

Les différentes jointures dans LINQ

Dans les exemples suivants, nous utiliserons les exemples suivants:

```
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,
  },
};
```

JOINTURE INTERNE

Syntaxe de requête

```
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();
```

Syntaxe de la méthode

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

```
                                    })
                    .ToList();
```

Résultat:

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

JOINTURE EXTERNE GAUCHE

```
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();
```

Résultat:

```
{ 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();
```

Résultat:

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

GROUPE REJOIGNEZ

```
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();
```

Les `Orders` propriétés contiennent désormais un `IEnumerable<Order>` avec toutes les commandes liées.

Résultat:

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

Comment adhérer à plusieurs conditions

Lorsque vous vous joignez à une seule condition, vous pouvez utiliser:

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

Lorsque vous vous joignez à plusieurs, utilisez:

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

Assurez-vous que les deux objets anonymes ont les mêmes propriétés, et dans VB.NET, ils doivent être marqués comme étant `Key` , bien que VB.NET autorise plusieurs clauses `Equals` séparées par `And` :

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

Syntaxe de requête et syntaxe de méthode

La syntaxe et la syntaxe de la requête sont identiques sur le plan sémantique, mais de nombreuses personnes trouvent la syntaxe de la requête plus simple et plus facile à lire. Disons que nous devons récupérer tous les articles, même ordonnés, dans un ordre croissant à partir d'une collection de nombres.

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)
```

N'oubliez pas que certaines requêtes **doivent** être exprimées sous forme d'appels de méthode. Par exemple, vous devez utiliser un appel de méthode pour exprimer une requête qui extrait le nombre d'éléments correspondant à une condition spécifiée. Vous devez également utiliser un appel de méthode pour une requête qui récupère l'élément qui a la valeur maximale dans une séquence source. Cela pourrait donc être un avantage d'utiliser la syntaxe de la méthode pour rendre le code plus cohérent. Cependant, vous pouvez bien sûr toujours appliquer la méthode après un appel de syntaxe de requête:

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();
```

Méthodes LINQ et IEnumerable vs IQueryable

Les méthodes d'extension LINQ sur `IEnumerable<T>` utilisent des méthodes réelles [1] , qu'il s'agisse de méthodes anonymes:

```
//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);
```

ou méthodes nommées (méthodes explicitement définies dans le cadre d'une 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
```

En théorie, il est possible d' analyser l'IL de la méthode , de déterminer ce que la méthode tente de faire et d'appliquer la logique de cette méthode à toute source de données sous-jacente, et pas seulement aux objets en mémoire. Mais analyser l'IL n'est pas pour les faibles de cœur.

Heureusement, .NET fournit l'interface `IQueryable<T>` et les méthodes d'extension de `System.Linq.Queryable` pour ce scénario. Ces méthodes d'extension utilisent un arbre d'expression - une structure de données représentant du code - au lieu d'une méthode réelle, que le fournisseur LINQ peut ensuite analyser [2] et convertir en un format plus approprié pour interroger la source de données sous-jacente. Par exemple:

```
//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 (par exemple) cette requête concerne une base de données SQL, le fournisseur peut convertir cette expression en l'instruction SQL suivante:

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

et l'exécuter avec la source de données.

En revanche, si la requête est associée à une API REST, le fournisseur peut convertir la même expression en appel API:

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

Il y a deux avantages principaux à adapter une demande de données basée sur une expression (au lieu de charger toute la collection en mémoire et d'interroger localement):

- La source de données sous-jacente peut souvent interroger plus efficacement. Par exemple, il peut très bien y avoir un index sur `LastName` . Le chargement des objets dans la mémoire locale et l'interrogation en mémoire perd cette efficacité.
- Les données peuvent être mises en forme et réduites avant d'être transférées. Dans ce cas, la base de données / service Web doit uniquement renvoyer les données correspondantes, par opposition à l'ensemble des personnes disponibles à partir de la source de données.

Remarques

1. Techniquement, ils ne prennent pas de méthodes, mais délèguent plutôt des instances qui pointent vers des méthodes . Cependant, cette distinction est sans importance ici.
2. C'est la raison des erreurs comme " *LINQ to Entities ne reconnaît pas la méthode" System.String ToString () ", et cette méthode ne peut pas être traduite en une expression de magasin.* ". Le fournisseur LINQ (dans ce cas, le fournisseur Entity Framework) ne sait pas analyser et traduire un appel à `ToString` en SQL équivalent.

Lire Démarrer avec linq en ligne: https://riptutorial.com/fr/linq/topic/842/demarrer-avec-linq

Chapitre 2: Linq utilisant Take while et Skip While

Introduction

Take, Skip, TakeWhile et SkipWhile sont tous des opérateurs de partitionnement puisqu'ils obtiennent une section d'une séquence d'entrée déterminée par une condition donnée. Parlons de ces opérateurs

Examples

Prendre méthode

La méthode Take Prend les éléments à une position spécifiée à partir du premier élément d'une séquence. **Signature de Take:**

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

Exemple:

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

Sortie:

Le résultat est 1,5,8,4 et 9 pour obtenir cinq éléments.

Méthode Skip

Saut des éléments à une position spécifiée à partir du premier élément d'une séquence.

Signature de Skip:

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

Exemple

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

Sortie: Le résultat est 3,6,7,2 et 0 pour obtenir l'élément.

TakeWhile ():

Retourne les éléments de la collection donnée jusqu'à ce que la condition spécifiée soit vraie. Si le premier élément lui-même ne satisfait pas la condition, il renvoie alors une collection vide.

Signature de TakeWhile ():

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

Une autre signature de surcharge:

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

Exemple I:

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

Sortie:

Il reviendra des résultats 1,5,8 et 4

Exemple II:

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

Sortie:

Il renverra de l'élément 1,2,3 et 4

SkipWhile ()

Ignore les éléments en fonction d'une condition jusqu'à ce qu'un élément ne réponde pas à la condition. Si le premier élément lui-même ne satisfait pas la condition, il ignore alors 0 élément et renvoie tous les éléments de la séquence.

Signature de SkipWhile ():

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

Une autre signature de surcharge:

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

Exemple I:

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

Sortie:

Il retournera de l'élément 9,3,6,7,2 et 0.

Exemple II:

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

Sortie:

Il renverra de l'élément 1,9,3,6,7,2 et 0.

Lire Linq utilisant Take while et Skip While en ligne: https://riptutorial.com/fr/linq/topic/10810/linq-utilisant-take-while-et-skip-while

Chapitre 3: Modes d'exécution de la méthode - immédiat, streaming différé, non-streaming différé

Examples

Exécution différée vs exécution immédiate

Certaines méthodes LINQ renvoient un objet de requête. Cet objet ne contient pas les résultats de la requête; au lieu de cela, il a toutes les informations nécessaires pour générer ces résultats:

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

La requête contient un appel à `Console.Write` , mais rien n'a été généré sur la console. En effet, la requête n'a pas encore été exécutée et la fonction passée à `Select` n'a donc jamais été évaluée. Ceci est connu sous le nom d' **exécution différée** - l' **exécution de** la requête est retardée jusqu'à un moment ultérieur.

D'autres méthodes LINQ forcent **l'exécution immédiate** de la requête. Ces méthodes exécutent la requête et génèrent ses valeurs:

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

À ce stade, la fonction passée dans `Select` sera évaluée pour chaque valeur de la liste d'origine, et les éléments suivants seront envoyés à la console:

    1 2 3 4 5

En général, les méthodes LINQ qui renvoient une seule valeur (telle que `Max` ou `Count`) ou qui renvoient un objet `ToList` réellement les valeurs (telles que `ToList` ou `ToDictionary`) s'exécutent immédiatement.

Les méthodes qui renvoient un `IEnumerable<T>` ou `IQueryable<T>` retournent l'objet de requête et permettent de reporter l'exécution à un stade ultérieur.

Si une méthode LINQ particulière oblige une requête à s'exécuter immédiatement ou non, vous pouvez la trouver sur MSDN- C # ou VB.NET .

Mode de diffusion (évaluation différée) vs mode sans diffusion (évaluation

rapide)

Parmi les méthodes LINQ qui utilisent l'exécution différée, certaines nécessitent une seule valeur à évaluer à la fois. Le code suivant:

```
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}");
}
```

va sortir:

> 3
> foreach valeur d'itération: 3
> 5
> foreach valeur d'itération: 5
> 1
> foreach valeur d'itération: 1
> 2
> foreach valeur d'itération: 2

car la fonction passée à `Select` est évaluée à chaque itération de `foreach` . Ceci est connu comme **mode de diffusion** ou **évaluation différée** .

D'autres méthodes LINQ (opérateurs de tri et de regroupement) exigent que *toutes* les valeurs soient évaluées avant de pouvoir renvoyer *une* valeur:

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

va sortir:

> 3
> 5
> 1
> 2
> foreach valeur d'itération: 1
> foreach valeur d'itération: 2
> foreach valeur d'itération: 3
> foreach valeur d'itération: 5

Dans ce cas, parce que les valeurs doivent être générées dans `foreach` dans l'ordre croissant, tous les éléments doivent d'abord être évalués, afin de déterminer lequel est le plus petit et le plus petit suivant, et ainsi de suite. Ceci est connu sous le nom de **mode non-streaming** ou **évaluation avide** .

Si une méthode LINQ particulière utilise le mode streaming ou non, peut être trouvée sur MSDN- C # ou VB.NET .

Avantages de l'exécution différée - création de requêtes

L'exécution différée permet de combiner différentes opérations pour générer la requête finale, avant d'évaluer les valeurs:

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

Si nous exécutons la requête à ce stade:

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

nous aurions la sortie suivante:

    2 2 3 4 6 9

Mais nous pouvons modifier la requête en ajoutant plus d'opérateurs:

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

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

Sortie:

    20 20 40 60

Avantages de l'exécution différée - interrogation des données actuelles

Avec l'exécution différée, si les données à interroger sont modifiées, l'objet de requête utilise les données au moment de l'exécution, pas au moment de la définition.

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

Si nous exécutons la requête à ce stade avec une méthode immédiate ou `foreach` , la requête

fonctionnera sur la liste des nombres pairs.

Cependant, si nous modifions les valeurs dans la liste:

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

ou même si on attribue une nouvelle liste aux `data` :

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

puis exécutez la requête, la requête fonctionnera sur la nouvelle valeur des `data` :

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

et affichera les informations suivantes:

    1 9 25 49 81

Lire Modes d'exécution de la méthode - immédiat, streaming différé, non-streaming différé en ligne: https://riptutorial.com/fr/linq/topic/7102/modes-d-execution-de-la-methode---immediat--streaming-differe--non-streaming-differe

Chapitre 4: Opérateurs de requêtes standard

Remarques

Les requêtes Linq sont écrites à l'aide des opérateurs de requête standard (un ensemble de méthodes d'extension fonctionnant principalement sur les objets de type `IEnumerable<T>` et `IQueryable<T>`) ou utilisant les expressions de requête (qui sont converties en opérateur de requête standard au moment de la compilation) appels de méthode).

Les opérateurs de requêtes fournissent des fonctionnalités de requête, notamment le filtrage, la projection, l'agrégation, le tri, etc.

Examples

Opérations de concaténation

La concaténation fait référence à l'opération consistant à ajouter une séquence à une autre.

Concat

> Concatène deux séquences pour former une séquence.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// Not applicable.
```

Opérations de filtrage

Le filtrage fait référence aux opérations de restriction du jeu de résultats pour ne contenir que les éléments satisfaisant à une condition spécifiée.

Où

> Sélectionne les valeurs basées sur une fonction de prédicat.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// 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 type

> Sélectionne des valeurs en fonction de leur capacité à être converties en un type spécifié.

Syntaxe de la méthode

```
// OfType

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

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

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

Syntaxe de requête

```
// Not applicable.
```

Rejoindre les opérations

Une jointure de deux sources de données est l'association d'objets dans une source de données avec des objets partageant un attribut commun dans une autre source de données.

Joindre

> Joint deux séquences basées sur des fonctions de sélection de clé et extrait des paires de valeurs.

Syntaxe de la méthode

```
// 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" }
```

Syntaxe de requête

```
// 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

> Joint deux séquences en fonction des fonctions du sélecteur de touches et regroupe les correspondances résultantes pour chaque élément.

Syntaxe de la méthode

```
// 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" }
```

Syntaxe de requête

```
// 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" }
```

Zip *: français

> Applique une fonction spécifiée aux éléments correspondants de deux séquences, produisant une séquence des résultats.

```
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 |
```

Opérations de projection

La projection fait référence aux opérations de transformation d'un objet en une nouvelle forme.

Sélectionner

> Projets de valeurs basées sur une fonction de transformation.

Syntaxe de la méthode

```
// Select

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

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

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

Syntaxe de requête

```
// 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

> Projette des séquences de valeurs basées sur une fonction de transformation, puis les aplatit en une séquence.

Syntaxe de la méthode

```
// 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") }
```

Syntaxe de requête

```
// multiples from

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

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

Opérations de tri

Une opération de tri commande les éléments d'une séquence en fonction d'un ou de plusieurs attributs.

Commandé par

Trie les valeurs en ordre croissant.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// 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

Trie les valeurs par ordre décroissant.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// 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 }
```

Puis par

Effectue un tri secondaire dans l'ordre croissant.

Syntaxe de la méthode

```
// 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" }
```

Syntaxe de requête

```
// 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

Effectue un tri secondaire par ordre décroissant.

Syntaxe de la méthode

```
// 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" }
```

Syntaxe de requête

```
// 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" }
```

Sens inverse

Inverse l'ordre des éléments dans une collection.

Syntaxe de la méthode

```
// Reverse

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

var reversed = numbers.Reverse();

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

Syntaxe de requête

```
// Not applicable.
```

Opérations de conversion

Les opérations de conversion modifient le type des objets en entrée.

AsEnumerable

Renvoie l'entrée typée IEnumerable.

Syntaxe de la méthode

```
// AsEnumerable

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

var nums = numbers.AsEnumerable();

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

Syntaxe de requête

```
// Not applicable.
```

AsQueryable

Convertit un IEnumerable en un IQueryable.

Syntaxe de la méthode

```
// AsQueryable
```

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

var nums = numbers.AsQueryable();

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

Syntaxe de requête

```
// Not applicable.
```

Jeter

Convertit les éléments d'une collection en un type spécifié.

Syntaxe de la méthode

```
// Cast

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

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

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

Syntaxe de requête

```
// 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 type

Filtre les valeurs en fonction de leur capacité à être converties en un type spécifié.

Syntaxe de la méthode

```
// OfType

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

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

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

Syntaxe de requête

```
// Not applicable.
```

ToArray

Convertit une collection en un tableau.

Syntaxe de la méthode

```
// ToArray

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

int[] array = numbers.ToArray();

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

Syntaxe de requête

```
// Not applicable.
```

Lister

Convertit une collection en une liste.

Syntaxe de la méthode

```
// ToList

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

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

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

Syntaxe de requête

```
// Not applicable.
```

ToDictionary

Met des éléments dans un dictionnaire en fonction d'une fonction de sélection de clé.

Syntaxe de la méthode

```
// ToDictionary

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

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

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

Syntaxe de requête

```
// Not applicable.
```

Opérations d'agrégation

Les opérations d’agrégation calculent une valeur unique à partir d’une collection de valeurs.

Agrégat

Effectue une opération d'agrégation personnalisée sur les valeurs d'une collection.

Syntaxe de la méthode

```
// Aggregate

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

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

// product = 120
```

Syntaxe de requête

```
// Not applicable.
```

Moyenne

Calcule la valeur moyenne d'une collection de valeurs.

Syntaxe de la méthode

```
// Average

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

var average = numbers.Average();

// average = 3
```

Syntaxe de requête

```
// Not applicable.
```

Compter

Compte les éléments d'une collection, éventuellement uniquement les éléments satisfaisant une fonction de prédicat.

Syntaxe de la méthode

```
// Count

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

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

// count = 2
```

Syntaxe de requête

```
// Not applicable.
```

LongCount

> Compte les éléments d'une grande collection, éventuellement uniquement les éléments satisfaisant une fonction de prédicat.

Syntaxe de la méthode

```
// LongCount

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

long count = numbers.LongCount();

// count = 10
```

Syntaxe de requête

```
// Not applicable.
```

Max

> Détermine la valeur maximale dans une collection. Lève une exception si la collecte est vide.

Syntaxe de la méthode

```
// Max

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

var max = numbers.Max();

// max = 5
```

Syntaxe de requête

```
// Not applicable.
```

Min

> Détermine la valeur minimale dans une collection. Lève une exception si la collecte est vide.

Syntaxe de la méthode

```
// Min

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

var min = numbers.Min();

// min = 1
```

Syntaxe de requête

```
// Not applicable.
```

Min- / MaxOrDefault

> Contrairement à d'autres extensions LinQ, `Min()` et `Max()` n'ont pas de surcharge sans exceptions. Pour cela, `IEnumerable` doit être vérifié pour `Any()` avant d’appeler `Min()` ou `Max()`

```
// Max

var numbers = new int[] { };

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

// max = 0
```

Somme

> Calcule la somme des valeurs dans une collection.

Syntaxe de la méthode

```
// Sum

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

var sum = numbers.Sum();

// sum = 15
```

Syntaxe de requête

```
// Not applicable.
```

Opérations de quantification

Les opérations du quantificateur renvoient une valeur booléenne qui indique si certains ou tous les éléments d'une séquence satisfont à une condition.

Tout

Détermine si tous les éléments d'une séquence satisfont à une condition.

Syntaxe de la méthode

```
// All

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

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

// areLessThan10 = true
```

Syntaxe de requête

```
// Not applicable.
```

Tout

Détermine si des éléments d'une séquence satisfont à une condition.

Syntaxe de la méthode

```
// Any

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

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

// anyOneIsEven = true
```

Syntaxe de requête

```
// Not applicable.
```

Contient

Détermine si une séquence contient un élément spécifié.

Syntaxe de la méthode

```
// Contains

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

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

// appears = false
```

Syntaxe de requête

```
// Not applicable.
```

Opérations de regroupement

Le regroupement fait référence aux opérations consistant à mettre des données en groupes afin que les éléments de chaque groupe partagent un attribut commun.

Par groupe

Groupes d'éléments qui partagent un attribut commun.

Syntaxe de la méthode

```
// 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") }
```

Syntaxe de requête

```
// 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") }
```

Pour rechercher

Insère des éléments dans un dictionnaire un à plusieurs basé sur une fonction de sélection de clé.

Syntaxe de la méthode

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

Syntaxe de requête

```
// Not applicable.
```

Opérations de partition

Le partitionnement fait référence aux opérations consistant à diviser une séquence d'entrée en deux sections, sans réorganiser les éléments, puis à renvoyer l'une des sections.

Sauter

Saut des éléments à une position spécifiée dans une séquence.

Syntaxe de la méthode

```
// Skip

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

var skipped = numbers.Skip(3);

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

Syntaxe de requête

```
// Not applicable.
```

SkipWhile

Ignore les éléments en fonction d'une fonction de prédicat jusqu'à ce qu'un élément ne satisfasse pas la condition.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// Not applicable.
```

Prendre

Prend des éléments à une position spécifiée dans une séquence.

Syntaxe de la méthode

```
// Take

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

var taken = numbers.Take(3);

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

Syntaxe de requête

```
// Not applicable.
```

TakeWhile

Prend des éléments basés sur une fonction de prédicat jusqu'à ce qu'un élément ne satisfasse pas la condition.

Syntaxe de la méthode

```
// TakeWhile

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

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

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

Syntaxe de requête

```
// Not applicable.
```

Opérations de génération

Generation désigne la création d'une nouvelle séquence de valeurs.

DefaultIfEmpty

Remplace une collection vide par une collection singleton évaluée par défaut.

Syntaxe de la méthode

```
// DefaultIfEmpty

var nums = new int[0];

var numbers = nums.DefaultIfEmpty();

// numbers = { 0 }
```

Syntaxe de requête

```
// Not applicable.
```

Vide

Retourne une collection vide.

Syntaxe de la méthode

```
// Empty

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

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

Syntaxe de requête

```
// Not applicable.
```

Gamme

Génère une collection qui contient une séquence de nombres.

Syntaxe de la méthode

```
// Range

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

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

Syntaxe de requête

```
// Not applicable.
```

Répéter

Génère une collection qui contient une valeur répétée.

Syntaxe de la méthode

```
// Repeat

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

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

Syntaxe de requête

```
// Not applicable.
```

Définir les opérations

Les opérations d'ensemble font référence aux opérations de requête qui produisent un jeu de résultats basé sur la présence ou l'absence d'éléments équivalents au sein de collections (ou d'ensembles) identiques ou distinctes.

Distinct

Supprime les valeurs en double d'une collection.

Syntaxe de la méthode

```
// Distinct

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

var distinct = numbers.Distinct();

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

Syntaxe de requête

```
// Not applicable.
```

Sauf

Renvoie la différence de set, ce qui signifie les éléments d'une collection qui n'apparaissent pas dans une deuxième collection.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// Not applicable.
```

Couper

> Renvoie l'intersection définie, c'est-à-dire les éléments qui apparaissent dans chacune des deux collections.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// Not applicable.
```

---

syndicat

> Renvoie l'union définie, ce qui signifie des éléments uniques qui apparaissent dans l'une des deux collections.

Syntaxe de la méthode

```
// 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 }
```

Syntaxe de requête

```
// Not applicable.
```

Opérations d'égalité

Deux séquences dont les éléments correspondants sont égaux et qui ont le même nombre d'éléments sont considérées égales.

SéquenceEqual

Détermine si deux séquences sont égales en comparant les éléments par paires.

Syntaxe de la méthode

```
// 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
```

Syntaxe de requête

```
// Not Applicable.
```

Opérations d'élément

Les opérations d'élément renvoient un seul élément spécifique d'une séquence.

ElementAt

Renvoie l'élément à un index spécifié dans une collection.

Syntaxe de la méthode

```
// ElementAt

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

var str = strings.ElementAt(2);

// str = "two"
```

Syntaxe de requête

```
// Not Applicable.
```

ElementAtOrDefault

Retourne l'élément à un index spécifié dans une collection ou une valeur par défaut si l'index est hors limites.

Syntaxe de la méthode

```
// ElementAtOrDefault

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

var str = strings.ElementAtOrDefault(10);

// str = null
```

Syntaxe de requête

```
// Not Applicable.
```

Premier

Renvoie le premier élément d'une collection ou le premier élément qui satisfait à une condition.

Syntaxe de la méthode

```
// First

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

var first = strings.First();

// first = 1
```

Syntaxe de requête

```
// Not Applicable.
```

FirstOrDefault

Renvoie le premier élément d'une collection ou le premier élément qui satisfait à une condition. Renvoie une valeur par défaut si aucun élément de ce type n'existe.

Syntaxe de la méthode

```
// FirstOrDefault

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

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

// firstGreaterThanTen = 0
```

Syntaxe de requête

```
// Not Applicable.
```

Dernier

Renvoie le dernier élément d'une collection ou le dernier élément qui satisfait à une condition.

Syntaxe de la méthode

```
// Last

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

var last = strings.Last();

// last = 5
```

Syntaxe de requête

```
// Not Applicable.
```

LastOrDefault

Renvoie le dernier élément d'une collection ou le dernier élément qui satisfait à une condition. Renvoie une valeur par défaut si aucun élément de ce type n'existe.

Syntaxe de la méthode

```
// LastOrDefault

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

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

// lastGreaterThanTen = 0
```

Syntaxe de requête

```
// Not Applicable.
```

Unique

Retourne le seul élément d'une collection ou le seul élément qui satisfait à une condition.

Syntaxe de la méthode

```
// Single

var numbers = new int[] { 1 };

var single = strings.Single();

// single = 1
```

Syntaxe de requête

```
// Not Applicable.
```

SingleOrDefault

> Retourne le seul élément d'une collection ou le seul élément qui satisfait à une condition. Renvoie une valeur par défaut si aucun élément n'existe ou si la collection ne contient pas exactement un élément.

Syntaxe de la méthode

```
// SingleOrDefault

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

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

// singleGreaterThanFour = 5
```

Syntaxe de requête

```
// Not Applicable.
```

Lire Opérateurs de requêtes standard en ligne: https://riptutorial.com/fr/linq/topic/2535/operateurs-de-requetes-standard

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec 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 utilisant Take while et Skip While	kari kalan
3	Modes d'exécution de la méthode - immédiat, streaming différé, non-streaming différé	Colin Young, mmushtaq, Travis J, Zev Spitz
4	Opérateurs de requêtes standard	Arturo Menchaca, James Cockayne, Mark Hurd, Toxantron