Introduction aux requêtes LINQ (C #)

07/20/2015 • 5 minutes pour lire • +8

Dans cet article

Une *requête* est une expression qui extrait des données d'une source de données. Les requêtes sont généralement exprimées dans un langage de requête spécialisé. Différents langages ont été développés au fil du temps pour les différents types de sources de données, par exemple SQL pour les bases de données relationnelles et XQuery pour XML. Par conséquent, les développeurs ont dû apprendre un nouveau langage de requête pour chaque type de source de données ou de format de données qu'ils doivent prendre en charge. LINQ simplifie cette situation en offrant un modèle cohérent pour travailler avec des données provenant de divers types de sources de données et de formats. Dans une requête LINQ, vous travaillez toujours avec des objets. Vous utilisez les mêmes modèles de codage de base pour interroger et transformer des données dans des documents XML, des bases de données SQL, des jeux de données ADO.NET, des collections .NET et tout autre format pour lequel un fournisseur LINQ est disponible.

Trois parties d'une opération de requête

Toutes les opérations de requête LINQ consistent en trois actions distinctes:

1. Obtenir la source de données.
2. Créez la requête.
3. Exécutez la requête.

L'exemple suivant montre comment les trois parties d'une opération de requête sont exprimées en code source. L'exemple utilise un tableau d'entiers comme source de données pour plus de commodité. Cependant, les mêmes concepts s'appliquent

également à d'autres sources de données. Cet exemple est mentionné dans le reste de cette rubrique.

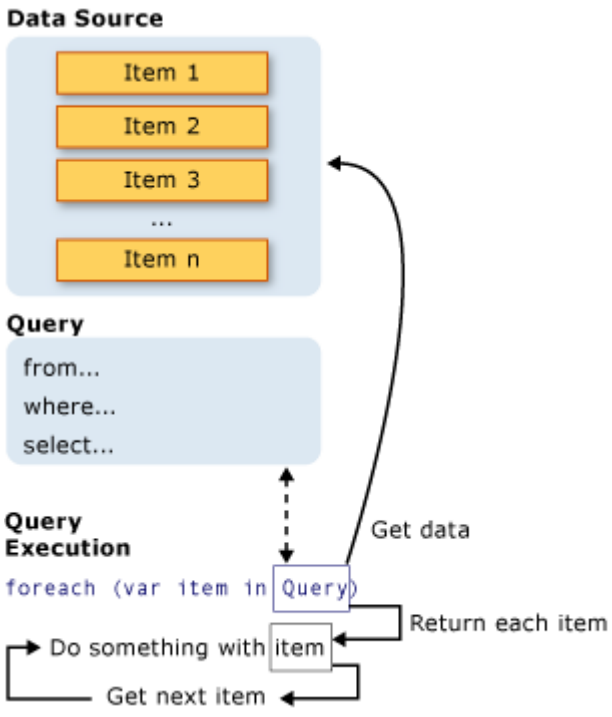
C # Copie

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

L'illustration suivante montre l'opération de requête complète. Dans LINQ, l'exécution de la requête est distincte de la requête elle-même; En d'autres termes, vous n'avez pas récupéré de données simplement en créant une variable de requête.



La source de données

Dans l'exemple précédent, la source de données étant un tableau, elle prend implicitement en charge l' interface générique IEnumerable <T> . Ce fait signifie qu'il peut être interrogé avec LINQ. Une requête est exécutée dans une `foreach` instruction et `foreach` nécessite IEnumerable ou IEnumerable <T> . Les types qui prennent en charge IEnumerable <T> ou une interface dérivée telle que le générique IQueryable <T> sont appelés *types interrogeables* .

Un type interrogeable ne nécessite aucune modification ni traitement spécial pour servir de source de données LINQ. Si les données source ne sont pas déjà en mémoire en tant que type interrogeable, le fournisseur LINQ doit les représenter en tant que telles. Par exemple, LINQ to XML charge un document XML dans un type XElement interrogeable :

C # Copie

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Avec LINQ to SQL, vous créez d'abord un mappage objet-relationnel au moment de la conception, manuellement ou à l'aide des outils LINQ to SQL dans Visual Studio dans Visual Studio. Vous écrivez vos requêtes sur les objets et, au moment de l'exécution, LINQ to SQL gère la communication avec la base de données. Dans l'exemple suivant, `Customers` représente une table spécifique dans la base de données et le type du résultat de la requête, IQueryable <T> , dérive de IEnumerable <T> .

C # Copie

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

For more information about how to create specific types of data sources, see the documentation for the various LINQ providers. However, the basic rule is very simple: a LINQ data source is any object that supports the generic IEnumerable<T> interface, or an interface that inherits from it.

> **Note**

Types such as **ArrayList** that support the non-generic **IEnumerable** interface can also be used as a LINQ data source. For more information, see **How to: Query an ArrayList with LINQ (C#)**.

The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. A query is stored in a query variable and initialized with a query expression. To make it easier to write queries, C# has introduced new query syntax.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where` and `select`. (If you are familiar with SQL, you will have noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. These and the other query clauses are discussed in detail in the LINQ Query Expressions section. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point. For more information about how queries are constructed behind the scenes, see Standard Query Operators Overview (C#).

> **Note**
>
> Queries can also be expressed by using method syntax. For more information, see **Query Syntax and Method Syntax in LINQ**.

Query Execution

Deferred Execution

As stated previously, the query variable itself only stores the query commands. The actual execution of the query is deferred until you iterate over the query variable in a `foreach` statement. This concept is referred to as *deferred execution* and is demonstrated in the following example:

C# Copy

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

The `foreach` statement is also where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it as often as you like. For example, you may have a database that is being updated continually by a separate application. In your application, you could create one query that retrieves the latest data, and you could execute it repeatedly at some interval to retrieve different results every time.

Forcing Immediate Execution

Queries that perform aggregation functions over a range of source elements must first iterate over those elements. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. Note also that these types of queries return a single value, not an `IEnumerable` collection. The following query returns a count of the even numbers in the source array:

C# Copy

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the ToList or ToArray methods.

C# Copy

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
```

```
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Vous pouvez également forcer l'exécution en plaçant la `foreach` boucle immédiatement après l'expression de la requête. Cependant, en appelant `ToList` ou en mettant en `ToArray` cache toutes les données d'un même objet de collection.

Voir également

- Démarrer avec LINQ en C #
- Procédure pas à pas: Écriture de requêtes en C #
- Expressions de requête LINQ
- foreach, dans
- Mots-clés de requête (LINQ)