CHAPTER 5

Introducing Language-Integrated Query (LINQ)

Emerging from a research project at Microsoft codenamed *Comega* (pronounced “see omega”), *Language-Integrated Query*, or LINQ as it’s now known, has been an integral part of the .NET Framework since version 3.5. Nothing else has advanced the evolution ADO.NET better than LINQ.

If you are building data-driven applications, you know the pain of fetching data from multiple data sources. At some point, you will need to emerge from your comfort zone and learn the native query languages of the data sources on which you are attempting to operate. This is a phenomenal task for developers given the overwhelming number of data sources from which applications written in .NET can fetch data.

Wouldn’t it be great if you could write the queries in .NET itself, and somehow they get magically transformed into the native query languages of the relevant data sources? With LINQ, you can do just that.

In this chapter, we will cover the following:

- How to write queries using LINQ.
- The different sources of data that can be queried using LINQ.
- The difference between enumerable and queryable types.
- How query operators function.
- How you can use LINQ to query ADO.NET data sources.
- How LINQ can leverage hyperthreading concepts to execute queries in parallel.
- How you can query Open Data (OData) Data Services using LINQ.

The ability to query data sources using one language is powerful, and it is used for building data-driven web sites. Hence, the concepts discussed in this chapter are important, and you will be using them moving forward through the remainder of this book.

**Writing Queries Using LINQ**

A LINQ expression has three basic parts. It is analogically equivalent to a database query expression. A great example is Transact-SQL (T-SQL), used to query and manipulate data in Microsoft SQL Server. If you are familiar with the T-SQL syntax, you know that a query to retrieve information is primarily formed using SELECT followed by the source using FROM, and it is then immediately followed by a condition using WHERE. There are, of course, additional keywords...
援助排序或分组。数据源是你创建的表，然后执行查询。在 LINQ 中，你有以下内容：

1. Enumerable 收集作为源。
2. 你使用 LINQ 查询语法构建查询（适用于多种语言，包括 VB 和 C#）。
3. 你执行查询并处理结果并将其放入变量。

让我们详细检查这些部分。

LINQ 数据源

LINQ 可以查询各种数据源。然而，这些源可以大致分为两个类别。第一类是内存中的数据源，其基本假设是数据已经加载到实现 IEnumerable 接口或其泛型等价物 IEnumerable<T> 的.NET 可枚举集合中。它们也可能实现 IQueryable 接口的派生类。LINQ 中实现任何这些接口的类型统称为“可查询类型”。

本质上，任何以某种形式（隐含或显式）实现 IEnumerable 的对象都可以被查询。这通常被称为“LINQ 对象”。

让我们探索 LINQ 的语法。

想象你有一个包含销售联系人的列表，并且你想获取来自加利福尼亚州的子集合。与 SQL 查询相似，从数据库中获取此信息的等效 SQL 查询将如下所示：

```
SELECT [Name], [Email] FROM Contacts WHERE [State] = 'CA'
```

如果你在商店中预加载了 Contacts 列表（List<Contact>），则 LINQ 查询将如下所示：

```
var result = from contact in contacts
              where contact.State == "CA"
              select contact;
```

要执行，您可以遍历 result 并对输出执行必要的操作。

```
foreach (var item in result)
{
    Console.WriteLine(item.Name);
}
```

**注** 查询在结果处理之前不会被执行。您可以通过调用查询结果上的聚合函数，例如 Count 或者通过使用 ToList 或 ToArray 延展方法来强制立即执行。

---

**Note**

Enumerable 收集在.NET 中通常实现 IEnumerable 接口或其泛型等价物 IEnumerable<T>。它们也可能实现 IQueryable 接口的派生类。Collectively, types that implement any of these interfaces are called *queryable types* in LINQ.

In essence, every object that implements IEnumerable in some form (implicitly or explicitly) is available to be queried. This is often referred to as *LINQ to Objects*.

Let's now explore the syntax of LINQ using a List. Imagine that you have a list of sales contacts and that you want a subset of contacts from the state of California. To draw similarities, an equivalent SQL query for fetching this information from the database would look like this:

```
SELECT [Name], [Email] FROM Contacts WHERE [State] = 'CA'
```

If you had preloaded all of the Contacts from the store into a List(List<Contact>), then the LINQ query would appear as follows:

```
var result = from contact in contacts
              where contact.State == "CA"
              select contact;
```

To execute, you can loop through the result and perform the necessary operations on the output.

```
foreach (var item in result)
{
    Console.WriteLine(item.Name);
}
```

**Note** The query is not executed until the results are processed. You can, however, force the immediate execution by calling aggregate functions like Count on the query result or by caching the result using the ToList or ToArray extension methods.