<https://www.cainiaoplus.com/linq/linq-tutorial.html>

**LINQ 教程**

語言集成查詢（LINQ）是.Net 3.5和Visual Studio 2008引入的功能強大的查詢語言。LINQ可與C＃或Visual Basic一起使用，以查詢不同的資料來源。

LINQ教程將使用從基礎到高級的主題來説明您學習LINQ語言。這些教程分為一系列相關主題，因此您從一個必須首先理解的主題開始，然後逐步地逐步學習LINQ的其他功能。LINQ教程包含了易於理解的解釋，實際示例，有用的技巧，資訊性注釋和要記住的要點。

這些教程是為想要逐步學習LINQ的初學者和專業人士而設計的。

**基礎條件**

必須具備.Net Framework3.5 / 4.5，C＃和Visual Studio的基本知識。

**面向群體**

本教程的目的是為那些熱衷於學習LINQ的初學者提供一個簡單和全面的LINQ理解。

**LINQ 簡介**

LINQ（語言集成查詢）是C＃和VB.NET中的統一查詢語法，用於從不同的源和格式檢索資料。它集成在C＃或VB中，從而消除了程式設計語言和資料庫之間的不匹配，並為不同類型的資料來源提供了單個查詢介面。

例如，SQL是一種結構化查詢語言，用於保存和檢索資料庫中的資料。同樣，LINQ是C＃和VB.NET中內置的結構化查詢語法，用於從不同類型的資料來源（例如集合，ADO.Net DataSet，XML Docs，Web服務和MS SQL Server和其他資料庫）中檢索資料。

LINQ查詢將結果作為對象返回。它使您可以在結果集上使用物件導向的方法，而不必擔心將不同格式的結果轉換為物件。

下麵的示例演示一個簡單的LINQ查詢，該查詢從包含“ a”的陣列中獲取所有字串。

示例：LINQ查詢到陣列

// 資料來源

string[] names = {"Bill", "Steve", "James", "Mohan" };

// LINQ查詢

var myLinqQuery = from name in names

                where name.Contains('a')

                select name;

// 查詢執行

foreach(var name in myLinqQuery)

    Console.Write(name + " ");

在上面的示例中，字串陣列名稱稱是一個資料來源。以下是分配給變數 myLinqQuery 的LINQ查詢。

from name in names

where name.Contains('a')

select name;

上面的查詢使用LINQ的查詢語法。您將在“查詢語法”一章中瞭解更多資訊。

在執行LINQ查詢之前，您不會得到它的結果。LINQ查詢可以以多種方式執行，這裡我們使用foreach迴圈來執行存儲在myLinqQuery中的查詢。foreach迴圈對資料來源執行查詢並獲得結果，然後在結果集上反覆運算。

因此，每個LINQ查詢都必須查詢某種資料來源，無論它可以是陣列，集合，XML還是其他資料庫。編寫LINQ查詢後，必須執行它才能獲得結果。

在下一章中瞭解為什麼要使用LINQ。

# LINQ 優點

為了理解為什麼我們應該使用LINQ，讓我們看一些示例。假設您要從一系列Student對象中查找青少年學生的列表。

在C＃2.0之前，我們必須使用“ foreach”或“ for”迴圈遍歷集合以查找特定物件。例如，我們必須編寫以下代碼，以從年齡在12至20歲（青少年13至19歲）的一系列學生中查找所有學生對象：

示例：在C#1.0中，使用for迴圈從集合中查找元素

class Student

{

    public int StudentID { get; set; }

    public String StudentName { get; set; }

    public int Age { get; set; }

}

class Program

{

    static void Main(string[] args)

    {

        Student[] studentArray = {

            new Student() { StudentID = 1, StudentName = "John", Age = 18 },

            new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 },

            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 },

            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },

            new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 },

            new Student() { StudentID = 6, StudentName = "Chris",  Age = 17 },

            new Student() { StudentID = 7, StudentName = "Rob",Age = 19  },

        };

        Student[] students = new Student[10];

        int i = 0;

        foreach (Student std in studentArray)

        {

            if (std.Age > 12 && std.Age < 20)

            {

                students[i] = std;

                i++;

            }

        }

    }

}

for迴圈的使用很麻煩，不可維護和可讀性差。C#2.0引入了委託，可以用來處理這種情況，如下所示。

    示例：使用委託從C＃2.0中的集合中查找元素

delegate bool FindStudent(Student std);

class StudentExtension

{

    public static Student[] where(Student[] stdArray, FindStudent del)

    {

        int i=0;

        Student[] result = new Student[10];

        foreach (Student std in stdArray)

            if (del(std))

            {

                result[i] = std;

                i++;

            }

        return result;

    }

}

class Program

{

    static void Main(string[] args)

    {

        Student[] studentArray = {

            new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

            new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

            new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 } ,

            new Student() { StudentID = 6, StudentName = "Chris",  Age = 17 } ,

            new Student() { StudentID = 7, StudentName = "Rob",Age = 19  } ,

        };

        Student[] students = StudentExtension.where(studentArray, delegate(Student std){

                return std.Age > 12 && std.Age < 20;

            });

        }

    }

}

因此，使用C#2.0，您可以利用委託的優勢找到符合任何條件的學生。你不必使用for迴圈來查找使用不同標準的學生。例如，可以使用相同的委託函數查找StudentId為5或姓名為Bill的學生，如下所示：

Student[] students = StudentExtension.where(studentArray, delegate(Student std) {

        return std.StudentID == 5;

    });

//此外，請使用同一委託使用其他條件

Student[] students = StudentExtension.where(studentArray, delegate(Student std) {

        return std.StudentName == "Bill";

    });

C＃團隊認為他們仍然需要使代碼更加緊湊和易讀。 因此，他們在C＃3.0中引入了擴展方法，lambda運算式，運算式樹，匿名類型和查詢運算式。 您可以使用C＃3.0的這些功能（它們是LINQ的構建塊）來查詢不同類型的集合並在單個語句中獲取結果元素。

下面的示例演示如何使用帶有 lambda 運算式的 LINQ 查詢從 student 集合中查找特定的學生。

示例：LINQ

class Program

{    static void Main(string[] args)

    {

          Student[] studentArray = {

                    new Student() { StudentID = 1, StudentName = "John", age = 18 } ,

                    new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,

                    new Student() { StudentID = 3, StudentName = "Bill",  age = 25 } ,

                    new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,

                    new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,

                    new Student() { StudentID = 6, StudentName = "Chris",  age = 17 } ,

                    new Student() { StudentID = 7, StudentName = "Rob",age = 19  } ,

           };

         // 使用LINQ查找青少年學生

         Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();

        // 使用LINQ查找名字為Bill的第一位學生

        Student bill = studentArray.Where(s => s.StudentName == "Bill").FirstOrDefault();

        // 使用LINQ查找StudentID為5的學生

        Student student5 = studentArray.Where(s => s.StudentID == 5).FirstOrDefault();

    }

}

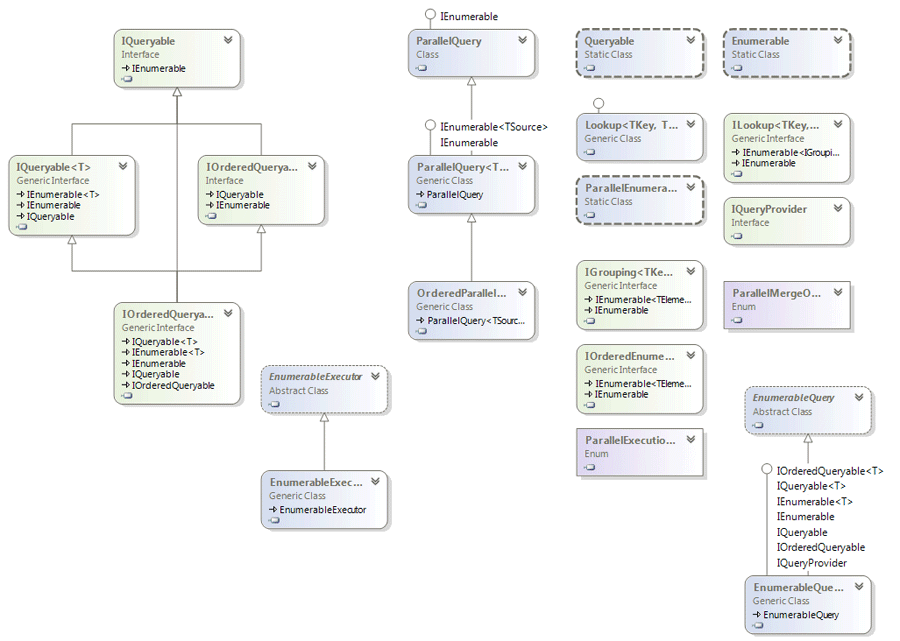
如上例所示，我們在單個語句中使用LINQ運運算元和lambda運算式指定了不同的條件。因此，LINQ使代碼更緊湊和可讀性強，並且還可以用於查詢不同的資料來源。例如，如果您在資料庫中有一個學生表，而不是上面的學生物件陣列，則仍可以使用相同的查詢使用實體框架查找學生。

## LINQ的優勢

* **Familiar language（熟悉的語言）:**開發人員不必為每種類型的資料來源或資料格式學習新的查詢語言。
* **Less coding（更少的代碼）:**與更傳統的方法相比，它減少了要編寫的代碼量。
* **Readable code（代碼可讀性）:**LINQ使代碼更具可讀性，因此其他開發人員可以輕鬆地理解和維護它。
* **Standardized way of querying multiple data sources（查詢多個資料來源的標準化方法）:**相同的LINQ語法可用於查詢多個資料來源。
* **Compile time safety of queries（查詢的編譯時安全性）:**它在編譯時提供物件的類型檢查。
* **IntelliSense Support（智慧感知支持）:**LINQ為通用集合提供了IntelliSense。
* **Shaping data（數據形狀）:**您可以以不同形狀檢索資料。

**LINQ API（.Net）**

我們可以為實現IEnumerable <T>或 IQueryable <T>介面的類編寫LINQ查詢。*System.Linq的*命名空間包括下列類和介面要求對LINQ查詢。

LINQ API

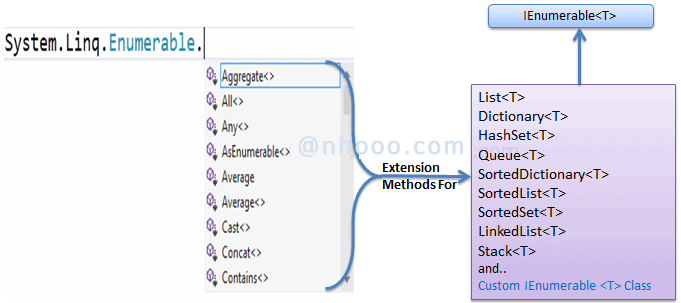
 在Visual Studio中添加新類時，默認包含 System.Linq 命名空間。。

LINQ查詢對實現IEnumerable或IQueryable介面的類使用擴展方法。Enumerable和Queryable是兩個靜態類，它們包含編寫LINQ查詢的擴展方法。

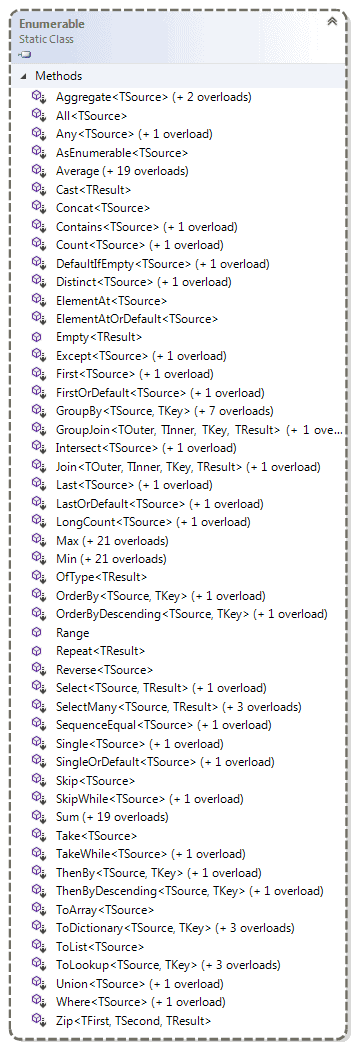
**可枚舉類(Enumerable)**

Enumerable類包括用於實現IEnumerable<T>介面的類的擴展方法，例如，所有內置集合類都實現了IEnumerable<T>介面，因此我們可以編寫LINQ查詢來從內置集合中檢索資料。

下圖顯示了Enumerable類中包含的擴展方法，可以與C＃或VB.Net中的泛型集合一起使用。



下圖顯示了Enumerable該類中所有可用的擴展方法。



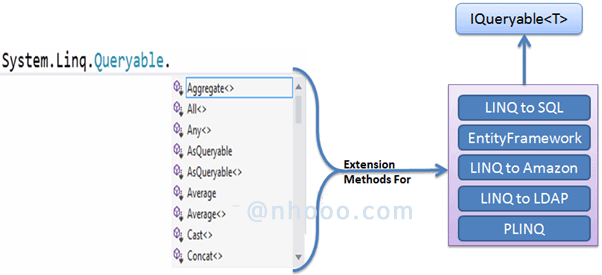
Enumerable 類

**可查詢(Queryable)**

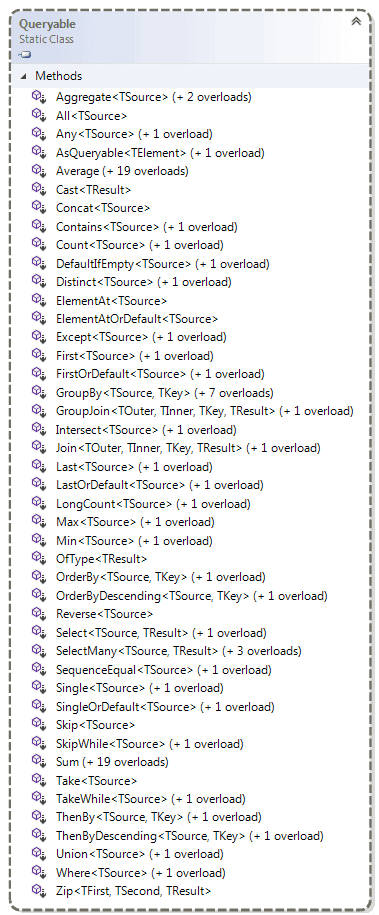
Queryable類包含用於實現成員“> IQueryable <t>介面的類的擴展方法。該IQueryable<T>介面用於提供針對已知資料類型的特定資料來源的查詢功能，例如，Entity Framework api實現了IQueryable<T>針對通過底層資料庫（例如MS SQL Server）支援LINQ查詢。

此外，還有一些API可用於訪問協力廠商資料。例如，LINQ to Amazon提供了將LINQ與Amazon Web服務結合使用以搜索書籍和其他物品的功能。這可以通過IQueryable為Amazon實現介面來實現。

下圖顯示了Queryable該類中可用的擴展方法，可以與各種本機或協力廠商資料提供程式一起使用。



下圖顯示了Queryable該類中可用的擴展方法。



Queryable 類

**要記住的要點**

1. 使用 System.LINQ 命名空間來使用 LINQ。
2. LINQ api包括兩個主要的靜態類Enumerable 和 Queryable。
3. 靜態Enumerable類包括用於實現IEnumerable <T>介面的類的擴展方法。
4. IEnumerable <T>集合的類型是記憶體中的集合，例如List，Dictionary，SortedList，Queue，HashSet，LinkedList。
5. 靜態Queryable類包括用於實現IQueryable <T>介面的類的擴展方法。
6. 遠端查詢提供程式實現了例如Linq-to-SQL，LINQ-to-Amazon等。

**LINQ 查詢語法**

有兩種將LINQ查詢寫入 IEnumerable 集合 或 IQueryable資料來源的基本方法。

1. 查詢語法或查詢運算式語法
2. 方法語法或方法擴展語法或連貫語法

**查詢語法**

查詢語法類似於資料庫的SQL（結構化查詢語言）。它在C＃或VB代碼中定義。

LINQ查詢語法：

from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>

<Standard Query Operators> <lambda expression>

<select or groupBy operator> <result formation>

LINQ查詢語法以from關鍵字開頭，以select關鍵字結尾。下面是一個示例LINQ查詢，該查詢返回一個字串集合，其中包含一個單詞“ Tutorials”。

示例：C＃中的LINQ查詢語法

// 字串集合

IList<string> stringList = new List<string>() {

    "C# Tutorials",

    "VB.NET Tutorials",

    "Learn C++",

    "MVC Tutorials" ,

    "Java"

};

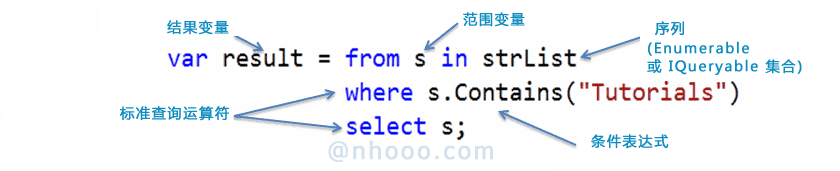
// LINQ查詢語法

var result = from s in stringList

            where s.Contains("Tutorials")

            select s;

下圖顯示了LINQ查詢語法的結構。

LINQ查詢語法

查詢語法以 From 子句開頭，後跟 Range 變數。From 子句的結構類似於“ From rangeVariableName in i enumerablecollection”。在英語中，這意味著，從集合中的每個物件。它類似於 foreach 迴圈:foreach(Student s in studentList)。

在FROM子句之後，可以使用不同的標準查詢運運算元來過濾，分組和聯接集合中的元素。LINQ中大約有50個標準查詢運運算元。在上圖中，我們使用了“ where”運運算元（又稱子句），後跟一個條件。通常使用lambda運算式來表達此條件。

LINQ查詢語法始終以Select或Group子句結尾。Select子句用於整形資料。您可以按原樣選擇整個物件，也可以僅選擇某些屬性。在上面的示例中，我們選擇了每個結果字串元素。

在下面的示例中，我們使用LINQ查詢語法從Student集合（序列）中找出青少年學生。

示例：C＃中的LINQ查詢語法

// 學生集合

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

// LINQ查詢語法找出青少年學生

var teenAgerStudent = from s in studentList

                      where s.Age > 12 && s.Age < 20

                      select s;

示例：VB.Net中的LINQ查詢語法

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

    }

// LINQ查詢語法找出青少年學生

Dim teenAgerStudents As IList(Of Student) = (From s In studentList \_

                                            Where s.Age > 12 And s.Age < 20 \_

                                            Select s).ToList()

**要記住的要點**

1. 顧名思義，**查詢語法**與SQL（結構查詢語言）語法相同。
2. 查詢語法以*from*子句開頭，可以以*Select*或*GroupBy*子句結尾。
3. 使用各種其他運運算元，例如過濾，聯接，分組，排序運運算元來構造所需的結果。
4. 隱式類型變數-var可用於保存LINQ查詢的結果。

# LINQ 方法語法

在上一節中，您已經瞭解了LINQ查詢語法。在這裡，您將瞭解方法語法。

方法語法（也稱為連貫語法）使用Enumerable 或 Queryable靜態類中包含的擴展方法，類似於您調用任何類的擴展方法的方式。

編譯器在編譯時將查詢語法轉換為方法語法。

以下是LINQ方法語法查詢示例，該查詢返回字串集合，其中包含單詞“ Tutorials”。

示例：C＃中的LINQ方法語法

// 字串集合

IList<string> stringList = new List<string>() {

    "C# Tutorials",

    "VB.NET Tutorials",

    "Learn C++",

    "MVC Tutorials" ,

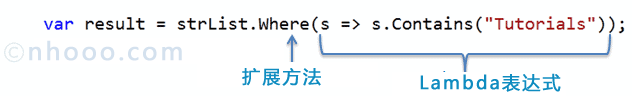
    "Java"

};

// LINQ查詢語法

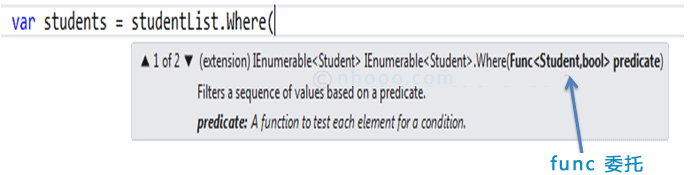
var result = stringList.Where(s => s.Contains("Tutorials"));

下圖說明瞭LINQ方法語法的結構。

LINQ方法語法結構

如上圖所示，方法語法包括擴展方法和 Lambda 運算式。在枚舉(Enumerable)類中定義的擴展方法 Where ()。

如果檢查Where擴展方法的簽名，就會發現Where方法接受一個 [predicate 委託](https://www.cainiaoplus.com/csharp/csharp-predicate.html) Func<Student,bool>。這意味著您可以傳遞任何接受Student物件作為輸入參數並返回布林值的委託函數，如下圖所示。lambda運算式用作Where子句中傳遞的委託。在下一節學習 Lambda 運算式。

Where 中的 Func 委託

下面的示例演示如何將LINQ方法語法查詢與IEnumerable <T>集合一起使用。

示例：C＃中的方法語法

// 學生集合

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

// LINQ方法語法找出青少年學生

var teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age < 20)

                                  .ToList<Student>();

示例：VB.Net中的方法語法

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

    }

// LINQ方法語法找出青少年學生

Dim teenAgerStudents As IList(Of Student) = studentList.Where(Function(s) s.Age > 12 And s.Age < 20)

                                                       .ToList()

## 要記住的要點

1. 顧名思義，**方法語法**就像調用擴展方法。
2. LINQ**方法語法**又稱Fluent語法（連貫語法），因為它允許一系列擴展方法調用。
3. 隱式類型變數-var可用於保存LINQ查詢的結果。

# LINQ Lambda 運算式

C＃3.0（.NET 3.5）引入了lambda運算式以及 LINQ。lambda運算式是使用某些特殊語法表示匿名方法的一種較短方法。

例如，採用以下匿名方法檢查學生是否為青少年：

示例：C＃中的匿名方法

delegate(Student s) { return s.Age > 12 && s.Age < 20; };

示例：VB.Net中的匿名方法

Dim isStudentTeenAger = Function(s As Student) As Boolean

                                    Return s.Age > 12 And s.Age < 20

                        End Function

可以使用C＃和VB.Net中的Lambda運算式來表示上述匿名方法，如下所示：

示例：C＃中的Lambda運算式

s => s.Age > 12 && s.Age < 20

示例：VB.Net中的Lambda運算式

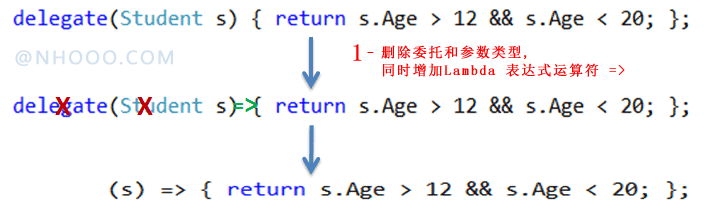
Function(s) s.Age  > 12 And s.Age < 20

讓我們看看lambda運算式是如何從以下匿名方法演變而來的。

示例：C＃中的匿名方法

delegate(Student s) { return s.Age > 12 && s.Age < 20; };

Lambda運算式是通過首先刪除委託關鍵字和參數類型並添加lambda運運算元=>從匿名方法演變而來的。

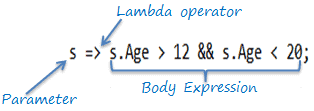
**來自匿名方法的Lambda運算式**

上面的lambda運算式絕對有效，但是如果我們只有一個返回值的語句，則不需要花括弧，return和分號。因此我們可以刪除它。

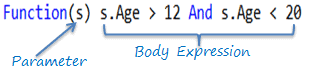
另外，如果只有一個參數，則可以刪除括弧()。

**來自匿名方法的Lambda運算式**

因此，我們得到了lambda運算式：s => s.Age > 12 && s.Age < 20 其中**s**是參數，**=>**是lambda運運算元，**s.Age > 12 && s.Age < 20**是運算式主體：

**C＃中的Lambda運算式結構**

我們在VB.Net中獲得lambda運算式的方式可以如下編寫：

**VB.Net中的Lambda運算式結構**

lambda運算式的調用方式與使用()的委託調用方式相同。

注意：VB.Net不支持lambda運運算元=>

## 具有多個參數的Lambda運算式

如果您需要傳遞多個參數，則可以將參數括在括弧中，如下所示：

    示例：在Lambda運算式C＃中指定多個參數

(s, youngAge) => s.Age >= youngage;

如果覺得參數表達不清，您還可以給出每個參數的類型：

示例：指定參數類型

(Student s,int youngAge) => s.Age >= youngage;

示例：在Lambda運算式中指定多個參數VB.Net

Function(s, youngAge) s.Age >= youngAge

## 不帶參數的Lambda運算式

lambda運算式中不必至少有一個參數。lambda運算式也可以不帶任何參數來指定。

示例：不帶參數的Lambda運算式

() => Console.WriteLine("無參數 lambda 運算式")

## Lambda表達主體中的多個語句

如果要在主體中包含多個語句，可以將運算式用大括弧括起來：

示例：C# 多語句Lambda運算式

(s, youngAge) =>{

  Console.WriteLine("在主體中包含多個語句的Lambda運算式");

  Return s.Age >= youngAge;}

示例：VB.Net多語句Lambda運算式

Function(s , youngAge)

    Console.WriteLine("在主體中包含多個語句的Lambda運算式")

    Return s.Age >= youngAge

End Function

## 在Lambda表達主體中聲明區域變數

您可以在運算式主體中聲明一個變數，以在運算式主體中的任何地方使用它，如下所示：

示例：C＃中的Lambda運算式區域變數

s =>

{   int youngAge = 18;

    Console.WriteLine("在主體中包含多個語句的Lambda運算式");

    return s.Age >= youngAge;

}

示例：VB.Net Lambda運算式中的區域變數

Function(s)

        Dim youngAge As Integer = 18

        Console.WriteLine("在主體中包含多個語句的Lambda運算式")

        Return s.Age >= youngAge

End Function

Lambda運算式也可以分配給內置委託，例如Func，Action和Predicate。

## 將Lambda運算式分配給委託

可以將lambda運算式分配給Func<in T, out TResult>委託類型。Func委託中的最後一個參數類型是返回類型，其餘參數是輸入參數。[訪問C＃教程的Func委託部分以瞭解更多資訊。](https://www.cainiaoplus.com/csharp/csharp-func-delegate.html)

考慮以下lambda運算式，以瞭解學生是否是青少年。

示例：C＃ Lambda運算式分配給Func 委託

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

Student std = new Student() { age = 21 };

bool isTeen = isStudentTeenAger(std);// 返回false

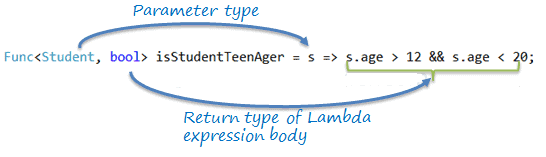
示例：VB.Net Lamda運算式分配給Func委託

Dim isStudentTeenAger As Func(Of Student, Boolean) = Function(s) s.Age > 12 And s.Age < 20

Dim stud As New Student With {.Age = 21}

Dim isTeen As Boolean = isStudentTeenAger(stud) // 返回 false

在上面的示例中，Func委託期望第一個輸入參數為Student類型，而返回類型為布林值。lambda運算式 s => s.age > 12 && s.age < 20滿足Func <Student，bool>委託要求，如下所示：

具有Lambda 運算式的Func委託

上面顯示的Func <>委託將變成一個如下所示的函數。

bool isStudentTeenAger(Student s)

{

    return s.Age > 12 && s.Age < 20;

}

## Action 委託

與Func委託不同，[Action委託](https://www.cainiaoplus.com/csharp/csharp-action-delegate.html)只能具有輸入參數。當您不需要從lambda運算式返回任何值時，請使用Action委託類型。

示例：C＃ Lamda運算式分配給動作委託

Action<Student> PrintStudentDetail = s => Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age);

Student std = new Student(){ StudentName = "Bill", Age=21};

PrintStudentDetail(std);//輸出: Name: Bill, Age: 21

示例：VB.Net Lamda運算式分配給動作委託

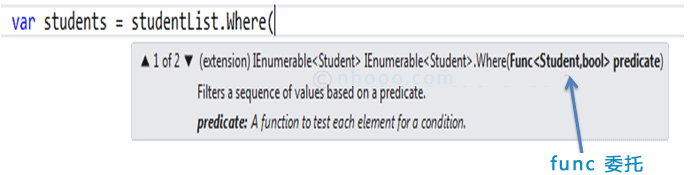
Dim printStudentDetail As Action(Of Student) = Sub(s) Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age)

Dim stud As New Student With {.StudentName = "Bill", .Age = 21}

printStudentDetail(stud)//輸出: Name: Bill, Age: 21

## LINQ查詢中的Lambda運算式

通常，lambda運算式與LINQ查詢一起使用。可枚舉的靜態類包括其中的擴展方法IEnumerable<T>wheres接受Func<TSource,bool>。因此，收集的Where()擴展方法IEnumerable<Student>需要通過Func<Student,bool>，如下所示：

Where擴展方法中的Func委託參數

因此，現在，您可以使用Where()如下所示的方法語法將分配給Func委託的lambda運算式傳遞給擴展方法：

示例：LINQ方法語法中的Func委託

IList<Student> studentList = new List<Student>(){...};

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

var teenStudents = studentList.Where(isStudentTeenAger).ToList<Student>();

示例：LINQ查詢語法中的Func委託

IList<Student> studentList = new List<Student>(){...};

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

var teenStudents = from s in studentList

                   where isStudentTeenAger(s)

                   select s;

您可以在VB.Net中遵循相同的方法來傳遞Func委託。

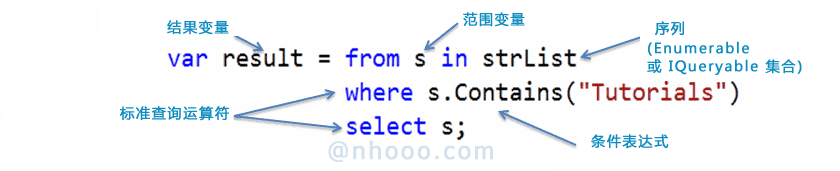
## 要記住的要點

1. Lambda運算式是表示匿名方法的簡寫方法。
2. Lambda運算式語法： 參數 => 運算式主體
3. Lambda運算式可以具有零參數。
4. Lambda運算式的括弧()中可以有多個參數。
5. Lambda運算式可以在大括弧{}中的主體運算式中包含多個語句。
6. Lambda運算式可以分配給Func，Action或Predicate委託。
7. 可以以類似於委託的方式調用Lambda Expression。

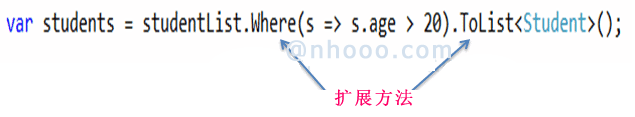
# LINQ 標準查詢運運算元

LINQ中的標準查詢運運算元實際上是 IEnumerable<T> and IQueryable<T>類型的擴展方法。它們在System.Linq.Enumerable和System.Linq.Queryable類中定義。LINQ中提供了50多個標準查詢運運算元，它們提供了不同的功能，例如過濾，排序，分組，聚合，串聯等。

## 查詢語法中的標準查詢運運算元

查詢語法中的標準查詢運運算元

## 方法語法中的標準查詢運運算元

方法語法中的標準查詢運運算元

查詢語法中的標準查詢運運算元在編譯時轉換為擴展方法。所以兩者都是一樣的。

可以根據標準查詢運運算元提供的功能對其進行分類。下表列出了標準查詢運運算元的所有分類：

| **類別** | **標準查詢運運算元** |
| --- | --- |
| 過濾 | Where, OfType |
| 排序 | OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse |
| 分組 | GroupBy, ToLookup |
| 聯合 | GroupJoin, Join |
| 投射 | Select, SelectMany |
| 聚合 | Aggregate, Average, Count, LongCount, Max, Min, Sum |
| 修飾 | All, Any, Contains |
| 元素 | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault |
| 集合 | Distinct, Except, Intersect, Union |
| 分區 | Skip, SkipWhile, Take, TakeWhile |
| 串聯 | Concat |
| 相等 | SequenceEqual |
| 範圍狀態 | DefaultEmpty, Empty, Range, Repeat |
| 轉換 | AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList |

在下一部分中瞭解每個標準查詢運運算元。

# LINQ 過濾運運算元 Where

LINQ中的過濾運運算元根據某些給定的標準過濾序列（集合）。

下表列出了LINQ中所有可用的過濾運運算元。

| **篩選運運算元** | **描述** |
| --- | --- |
| Where | 根據謂詞函數從集合中返回值。 |
| OfType | 根據指定類型返回集合中的值。 然而，它取決於它們是否能夠向指定類型轉換。 |

## Where

Where運運算元（Linq擴展方法）基於給定的條件運算式過濾集合並返回新集合。可以將標準指定為lambda運算式或Func委託類型。

**Where**擴展方法有以下兩個重載。兩種重載方法都接受Func委託類型參數。一個重載需要Func <TSource，bool>輸入參數，第二個重載方法需要Func <TSource，int，bool>輸入參數，其中int用於索引：

Where方法重載：

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

                                                  Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

                                                  Func<TSource, int, bool> predicate);

### 查詢語法中的Where子句

下面的查詢示例使用Where運運算元從給定的集合（序列）中篩選出青少年的學生。它使用lambda運算式作為謂詞函數。

示例：C＃ Where子句-LINQ查詢語法

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

var filteredResult = from s in studentList

                    where s.Age > 12 && s.Age < 20

                    select s.StudentName;

示例：Where子句-VB.Net中的LINQ查詢語法

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

    }

Dim filteredResult = From s In studentList

                     Where s.Age > 12 And s.Age < 20

                     Select s.StudentName

在上面的示例中，filteredResult 將在查詢執行後包括以下學生。

John

Bill

Ron

在上面的示例查詢中，lambda運算式主體 **s.Age > 12 && s.Age < 20** 作為評估集合中每個學生的謂詞函數傳遞。**Func<TSource, bool>**

另外，您還可以將Func類型委託與匿名方法一起使用，作為如下的謂詞函數進行傳遞（輸出是相同的）：

示例：Where子句

Func<Student,bool> isTeenAger = delegate(Student s) {

                                    return s.Age > 12 && s.Age < 20;

                                };

var filteredResult = from s in studentList

                     where isTeenAger(s)

                     select s;

你也可以通過Where()方法的重載調用任何與Func形參匹配的方法。

示例：Where子句

public static void Main()

{

    var filteredResult = from s in studentList

                         where isTeenAger(s)

                         select s;

}

public static bool IsTeenAger(Student stud)

{

    return stud.Age > 12 && stud.Age < 20;

}

### 方法語法中的where擴展方法

與查詢語法不同，您需要將整個lambda運算式作為謂詞函數傳遞，而不僅僅是LINQ方法語法中的運算式主體。

示例：C＃中方法語法的位置

var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);

示例：VB.Net中方法語法的位置

Dim filteredResult = studentList.Where(Function(s) s.Age > 12 And s.Age < 20 )

如上所述，**Where**擴展方法還具有第二重載，其包括集合中當前元素的索引。如果需要，可以在邏輯中使用該索引。

以下示例使用Where子句過濾出集合中的奇數元素，僅返回偶數元素。請記住，索引從零開始。

示例：Linq-C＃中的where擴展方法

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

var filteredResult = studentList.Where((s, i) => {

            if(i % 2 ==  0) // 如果是偶數

                return true;

        return false;

    });

foreach (var std in filteredResult)

        Console.WriteLine(std.StudentName);

輸出：

John

Bill

Ron

### 多個where子句

您可以在單個 LINQ 查詢中多次調用 Where() 擴展方法。

示例：查詢語法C＃中的多個where子句

var filteredResult = from s in studentList

                        where s.Age > 12

                        where s.Age < 20

                        select s;

示例：C＃中的方法語法多個where子句

var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);

## 要記住的要點

1. **Where**用於根據給定標準過濾集合。
2. 其中擴展方法有兩種重載方法。使用第二個重載方法可以知道集合中當前元素的索引。
3. 方法語法需要Where擴展方法中的整個lambda運算式，而查詢語法只需要運算式體。
4. 在單個LINQ查詢中，多個**Where**擴展方法有效。

**LINQ 過濾運運算元 OfType**

OfType運運算元基於將集合中的元素強制轉換為指定類型的能力來過濾篩選集合。

**查詢語法中的OfType**

使用OfType運運算元根據每個元素的類型篩選上述集合

示例：C＃中的OfType運運算元

IList mixedList = new ArrayList();

mixedList.Add(0);

mixedList.Add("One");

mixedList.Add("Two");

mixedList.Add(3);

mixedList.Add(new Student() { StudentID = 1, StudentName = "Bill" });

var stringResult = from s in mixedList.OfType()

                select s;

var intResult = from s in mixedList.OfType()

                select s;

示例：VB.Net中的OfType運運算元：

Dim stringResult = From s In mixedList.OfType(Of String)()

上面的示例查詢將返回mixedList中類型為字串的專案。執行後，stringResult包含以下元素：

One

Two

0

3

Bill

**方法語法中的OfType**

您可以在linq方法語法中使用OfType <TResult>()擴展方法，如下所示。

示例：C＃中的OfType

var stringResult = mixedList.OfType<string>();

示例：VB.Net中的OfType

Dim stringResult = mixedList.OfType(Of String)

stringResult將包含以下元素。

One

Two

**要記住的要點**

1. **Where**操作符根據謂詞函數篩選集合。
2. **OfType**操作符根據給定的類型對集合進行篩選。
3. **Where**和 **OfType**擴展方法可以在一個LINQ查詢中被多次調用。

**LINQ 排序運運算元 OrderBy和OrderByDescending**

排序運運算元以昇冪或降冪排列集合的元素。LINQ包括以下排序運運算元。

| **運運算元** | **描述** |
| --- | --- |
| OrderBy | 根據指定的欄位按昇冪或降冪對集合中的元素進行排序。 |
| OrderByDescending | 根據指定的欄位元按降冪對集合進行排序。僅在方法語法中有效。 |
| ThenBy | 僅在方法語法中有效。用於按昇冪進行二次排序。 |
| ThenByDescending | 僅在方法語法中有效。用於按降冪進行二次排序。 |
| Reverse | 僅在方法語法中有效。按相反順序對集合排序。 |

**OrderBy**

orderderby按昇冪或降冪對集合的值進行排序。預設情況下，它按昇冪對集合進行排序，因為ascending關鍵字在這裡是可選的。使用降冪關鍵字對集合進行降冪排序。

示例：C＃中的查詢語法 OrderBy

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var orderByResult = from s in studentList

                   orderby s.StudentName

                   select s;

var orderByDescendingResult = from s in studentList

                   orderby s.StudentName descending

                   select s;

示例：VB.Net 查詢語法中的 OrderBy

Dim orderByResult = From s In studentList

                   Order By s.StudentName

                   Select s

Dim orderByDescendingResult = From s In studentList

                   Order By s.StudentName Descending

                   Select s

上例中的orderByResult在執行後將包含以下元素：

Bill

John

Ram

Ron

Steve

上例中的orderByDescendingResult在執行後將包含以下元素：

Steve

Ron

Ram

John

Bill

**方法語法中的OrderBy**

OrderBy擴展方法有兩個重載。OrderBy擴展方法的第一個重載接受Func委託類型參數。因此，您需要為要對集合進行排序的欄位傳遞lambda運算式。

OrderBy的第二個重載方法接受IComparer的物件以及Func委託類型，以使用自訂比較進行排序。

OrderBy重載方法：

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source,

            Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source,

            Func<TSource, TKey> keySelector,

            IComparer<TKey> comparer);

以下示例使用OrderBy擴展方法按StudentName的昇冪對studentList集合進行排序。

示例：C＃中方法語法的OrderBy

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);

示例：VB.Net中方法語法的OrderBy

Dim studentsInAscOrder = studentList.OrderBy(Function(s) s.StudentName)

方法語法不允許decending關鍵字對集合進行降冪排序。使用OrderByDecending()方法。

**OrderByDescending**

OrderByDescending以降冪對集合進行排序。

OrderByDescending僅對方法語法有效。它在查詢語法中無效，因為查詢語法使用昇冪和降冪屬性，如上所示。

示例：C＃ 的 OrderByDescending

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);

示例：VB.Net 的 OrderByDescending

Dim studentsInDescOrder = studentList.OrderByDescending(Function(s) s.StudentName)

上面示例中的結果在執行後將包含以下元素。

Steve

Ron

Ram

John

Bill

請注意，查詢語法不支援OrderByDescending。請改用decending關鍵字。

**多重排序**

您可以在用逗號分隔的多個欄位元上對集合進行排序。給定的集合將首先基於第一個欄位進行排序，然後如果兩個元素的第一個欄位元的值相同，則將使用第二個欄位進行排序，依此類推。

    示例：查詢語法C＃中的多重排序

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },

    new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }

};

var orderByResult = from s in studentList

                   orderby s.StudentName, s.Age

                   select new { s.StudentName, s.Age };

在上面的示例中，studentList集合包括兩個相同的StudentName，Ram。因此，現在，studentList將首先基於StudentName進行排序，然後根據年齡進行昇冪排列。因此，執行後orderByResult將包含以下元素

StudentName: Bill, Age: 25

StudentName: John, Age: 18

StudentName: Ram, Age: 18

StudentName: Ram, Age: 20

StudentName: Ron, Age: 19

StudentName: Steve, Age: 15

方法語法中的多重排序的工作方式不同。使用ThenBy或ThenByDecenting擴展方法進行二次排序。

**要記住的要點**

1. LINQ包括五個排序運運算元：OrderBy，OrderByDescending，ThenBy，ThenByDescending和Reverse
2. LINQ 查詢語法不支援 OrderByDescending，ThenBy，ThenByDescending 和 Reverse。它只支持“ Order By”子句的“ ascending”和“ descending”排序方向。
3. LINQ查詢語法支援多個以逗號分隔的排序欄位，而您必須使用ThenBy和ThenByDescending方法進行二次排序

**LINQ 排序運運算元 ThenBy和ThenByDescending**

ThenBy和ThenByDescending擴展方法用於對多個欄位排序。

OrderBy ()方法根據指定的欄位元按昇冪對集合進行排序。在 OrderBy 之後使用 ThenBy ()方法按昇冪對另一個欄位元上的集合進行排序。Linq 首先根據 OrderBy 方法指定的主欄位元對集合進行排序，然後根據 ThenBy 方法指定的輔助欄位按昇冪再次對結果集合進行排序。

以相同的方式，使用ThenByDescending方法以降冪應用二次排序。

下面的示例演示如何使用ThenBy和ThenByDescending方法進行第二級排序：

示例：ThenBy＆ThenByDescending

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },

    new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }

};

var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);

如您在上面的示例中所見，我們首先按排序studentList集合StudentName，然後按排序Age。因此，現在，thenByResult排序後將包含以下元素：

StudentName: Bill, Age: 25

StudentName: John, Age: 18

StudentName: Ram, Age: 18

StudentName: Ram, Age: 20

StudentName: Ron, Age: 19

StudentName: Steve, Age: 15

現在 bydescresult 將包含以下元素。請注意，年齡為20歲的 Ram 比年齡為18歲的 Ram 更早出現，因為它使用了 ThenByDescending 。

StudentName: Bill, Age: 25

StudentName: John, Age: 18

StudentName: Ram, Age: 20

StudentName: Ram, Age: 18

StudentName: Ron, Age: 19

StudentName: Steve, Age: 15

您可以在VB.Net中以相同的方式使用ThenBy和ThenByDescending方法，如下所示：

示例：ThenBy＆ThenByDescending VB.Net

Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)

                              .ThenBy(Function(s) s.Age)

Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)

                              .ThenByDescending(Function(s) s.Age)

**要記住的要點**

1. 預設情況下，OrderBy和ThenBy對集合進行昇冪排序。
2. thenBy或ThenByDescending用於方法語法中的第二級排序。
3. thenByDescending方法在另一個欄位上按降冪對集合進行排序。
4. ThenBy或ThenByDescending在查詢語法中不適用。
5. 通過使用逗號分隔欄位，在查詢語法中應用二級排序。

接下來瞭解有關分組運運算元的資訊。

**LINQ 分組運運算元 GroupBy和ToLookup**

分組運運算元的作用與SQL查詢的GroupBy子句相同。分組運運算元根據給定的鍵創建一組元素。該組包含在實現IGrouping <TKey，TSource>介面的特殊類型的集合中，其中TKey是鍵值，在其上已形成該組，TSource是與該分組鍵值匹配的元素的集合。

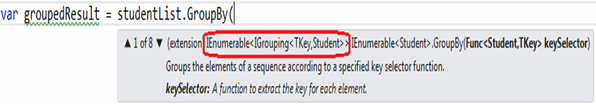
| **分組運運算元** | **描述** |
| --- | --- |
| GroupBy | GroupBy操作符根據某個鍵值返回元素組。每個組由 IGrouping<TKey，TElement>物件表示。 |
| ToLookup | ToLookup 與 GroupBy 相同; 唯一的區別是 GroupBy 的執行被延遲，而 ToLookup 的執行是立即的。 |

**GroupBy**

GroupBy運運算元基於某個鍵值從給定集合返回一組元素。每個組由IGrouping <TKey，TElement>物件表示。另外，GroupBy方法有8個重載方法，因此您可以根據需要在方法語法中使用適當的擴展方法。

LINQ查詢可以以GroupBy或Select子句結尾。

GroupBy運運算元的結果是組的集合。例如，GroupBy從Student集合返回IEnumerable <IGrouping <TKey，Student >>：

返回類型 GroupBy()

**查詢語法中的GroupBy**

下面的示例創建一組年齡相同的學生。年齡相同的學生將在同一集合中，每個分組的集合將具有一個金鑰和內部集合，其中金鑰將是年齡，內部集合將包括年齡與金鑰匹配的學生。

示例：GroupBy使用查詢語法C＃

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

    };

var groupedResult = from s in studentList

                    group s by s.Age;

//遍歷每組

foreach (var ageGroup in groupedResult)

{

    Console.WriteLine("Age Group: {0}", ageGroup .Key); //每組都有一個鑰匙

    foreach(Student s in ageGroup) // 每組都有內部收藏

        Console.WriteLine("Student Name: {0}", s.StudentName);

}

輸出：

AgeGroup: 18

StudentName: John

StudentName: Bill

AgeGroup: 21

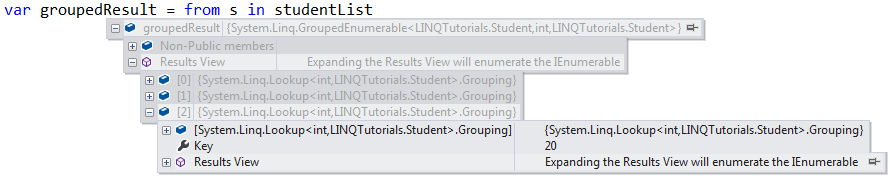
StudentName: Steve

StudentName: Abram

AgeGroup: 20

StudentName: Ram

如上例所示，您可以使用“ foreach”迴圈對組進行反覆運算，其中每個組都包含一個鍵和內部集合。下圖顯示了調試視圖中的結果。

通過鍵和內部集合對集合進行分組

如下所示，在VB.Net中，將“Into Group”與中的“Group By”子句一起使用。

示例：VB.Net中的GroupBy子句

Dim groupQuery = From s In studentList

                 Group By s.Age Into Group

For Each group In groupQuery

    Console.WriteLine("Age Group: {0}", group.Age) // 每個組都有關鍵屬性名稱

    For Each student In group.Group // 每組都有內部收藏

        Console.WriteLine("Student Name: {0}", student.StudentName)

    Next

Next

請注意，每個組都有一個執行該組的屬性名稱。在上面的示例中，我們使用Age來組成一個組，因此每個組將使用“ Age”屬性名稱而不是“ Key”作為屬性名稱。

輸出：

AgeGroup: 18

StudentName: John

StudentName: Bill

AgeGroup: 21

StudentName: Steve

StudentName: Abram

AgeGroup: 20

StudentName: Ram

**方法語法中的GroupBy**

GroupBy()擴展方法工作在方法的語法一樣。在GroupBy擴展方法中為鍵選擇器欄位名稱指定lambda運算式。

示例：C＃中 方法語法的GroupBy

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

    };

var groupedResult = studentList.GroupBy(s => s.Age);

foreach (var ageGroup in groupedResult)

{

    Console.WriteLine("Age Group: {0}", ageGroup.Key);  //每組都有一個鑰匙

    foreach(Student s in ageGroup)  //每個組都有一個內部集合

        Console.WriteLine("Student Name: {0}", s.StudentName);

}

示例：VB.Net GroupBy方法語法

Dim groupQuery = studentList.GroupBy(Function(s) s.Age)

For Each ageGroup In groupQuery

    Console.WriteLine("Age Group: {0}", ageGroup.Key)  //每組都有一個鑰匙

    For Each student In ageGroup.AsEnumerable()  //每個組都有一個內部集合

        Console.WriteLine("Student Name: {0}", student.StudentName)

    Next

Next

輸出：

AgeGroup: 18

StudentName: John

StudentName: Bill

AgeGroup: 21

StudentName: Steve

StudentName: Abram

AgeGroup: 20

StudentName: Ram

**ToLookup**

ToLookup與GroupBy相同；唯一的區別是GroupBy執行被推遲，而ToLookup執行是立即執行。另外，ToLookup僅適用於方法語法。**查詢語法不支援 ToLookup。**

示例：C＃中方法語法的 ToLookup

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

    };

var lookupResult = studentList.ToLookup(s => s.age);

foreach (var group in lookupResult)

{

    Console.WriteLine("Age Group: {0}", group.Key);  //每組都有一個鍵

    foreach(Student s in group)  //每個組都有一個內部集合

        Console.WriteLine("Student Name: {0}", s.StudentName);

}

示例：VB.Net中方法語法的ToLookup

Dim loopupResult = studentList.ToLookup(Function(s) s.Age)

**要記住的要點**

1. GroupBy＆ToLookup返回一個具有鍵和基於鍵欄位元值的內部集合的集合。
2. GroupBy 延遲執行，而 ToLookup 立即執行。
3. LINQ查詢語法可以以GroupBy或Select子句結尾。

**LINQ 聯結運運算元 Join**

聯結運運算元將兩個序列（集合）聯接並產生結果。

| **聯結運運算元** | **用法** |
| --- | --- |
| Join | Join 運運算元根據一個鍵連接兩個序列(集合)並返回結果序列。 |
| GroupJoin | GroupJoin 運運算元根據鍵連接兩個序列並返回序列組。它類似于 SQL 的左外聯接。 |

**Join**

Join運運算元對兩個集合（內部集合和外部集合）進行操作。它返回一個新集合，其中包含兩個集合中滿足指定運算式的元素。它與SQL的**內部(inner join)**聯接相同。

**Join方法語法**

Join擴展方法有兩個重載，如下所示。

Join重載方法：

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

            IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector,

            Func<TInner, TKey> innerKeySelector,

            Func<TOuter, TInner, TResult> resultSelector);

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

            IEnumerable<TInner> inner,

            Func<TOuter, TKey> outerKeySelector,

            Func<TInner, TKey> innerKeySelector,

            Func<TOuter, TInner, TResult> resultSelector,

            IEqualityComparer<TKey> comparer);

正如您在第一個重載中看到的，方法接受五個輸入參數（除了第一個“this”參數）：1）outer 2）inner 3）outerKeySelector 4）inner keyselector 5）resultSelector。

讓我們舉一個簡單的實例。下麵的示例連接兩個字串集合，並返回兩個集合中都包含匹配字串的新集合。

示例：C＃ join運運算元

IList<string> strList1 = new List<string>() {

    "One",

    "Two",

    "Three",

    "Four"

};

IList<string> strList2 = new List<string>() {

    "One",

    "Two",

    "Five",

    "Six"

};

var innerJoin = strList1.Join(strList2,

                      str1 => str1,

                      str2 => str2,

                      (str1, str2) => str1);

輸出結果:

One

Two

現在，讓我們瞭解使用下面的Student和Standard類的join方法，其中Student類包括與Standard類的StandardID相匹配的StandardID。

示例類

public class Student{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int StandardID { get; set; }

}

public class Standard{

    public int StandardID { get; set; }

    public string StandardName { get; set; }

}

下麵的示例演示LINQ Join查詢。

    示例：C＃ 聯接查詢

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", StandardID =1 },

    new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },

    new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },

    new Student() { StudentID = 4, StudentName = "Ram" , StandardID =2 },

    new Student() { StudentID = 5, StudentName = "Ron"  }

};

IList<Standard> standardList = new List<Standard>() {

    new Standard(){ StandardID = 1, StandardName="Standard 1"},

    new Standard(){ StandardID = 2, StandardName="Standard 2"},

    new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var innerJoin = studentList.Join(// 外序列

                      standardList,  // 內部序列

                      student => student.StandardID,    // externalKeySelector

                      standard => standard.StandardID,  // innerKeySelector

                      (student, standard) => new  // 結果選擇器

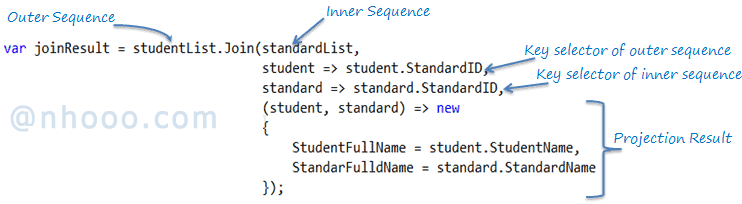
                                    {

                                        StudentName = student.StudentName,

                                        StandardName = standard.StandardName

                                    });

下圖說明瞭上面示例中的Join運運算元的各個部分。

Join 運運算元

在上面的聯接查詢示例中，studentList是外部序列，因為查詢從它開始。Join方法中的第一個參數用於指定內部序列，在上面的示例中該序列為standardList。Join方法的第二個和第三個參數用於指定一個欄位，該欄位的值應使用lambda運算式匹配，以便將元素包括在結果中。外部序列的鍵選擇器 student => student.StandardID指示StudentList的每個元素的標準ID欄位應該與內部序列 standard => standard.StandardID 的鍵匹配。如果兩個鍵欄位的值都匹配，則將該元素包括到結果中。

Join方法中的最後一個參數是用於表達結果的運算式。在上面的示例中，結果選擇器包括兩個序列的StudentName和StandardName屬性。

兩個序列（集合）的StandardID鍵必須匹配，否則該項將不包括在結果中。例如，Ron不與任何標準關聯，因此Ron不包含在結果集合中。上述示例中的innerJoinResult在執行後將包含以下元素：

John - Standard 1

Moin - Standard 1

Bill - Standard 2

Ram - Standard 2

下麵的示例演示VB.Net中方法語法中的Join運運算元。

**示例：VB.Net的Join運運算元**

Dim innerJoin = studentList.Join(standardList,

                                    Function(s) s.StandardID,

                                    Function(std) std.StandardID,

                                    Function(s, std) New With

                                    {

                                        .StudentName = s.StudentName,

                                        .StandardName = std.StandardName

                                    });

**聯接查詢語法**

查詢語法中的連接運運算元的工作原理與方法語法略有不同。它需要外部序列、內部序列、鍵選擇器和結果選擇器“on”關鍵字用於鍵選擇器，其中“equals”運運算元的左側是outerKeySelector，“equals”運運算元的右側是innerKeySelector。

語法：join查詢語法

from ... in outerSequence

        join ... in innerSequence

        on outerKey equals innerKey

        select ...

下面的查詢語法中的Join運運算元示例，如果Student.StandardID和Standard.StandardID匹配，則返回來自studentList和standardList的元素的集合。

**示例：C＃使用查詢語法join聯結運運算元**

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 13, StandardID =1 },

    new Student() { StudentID = 2, StudentName = "Moin",  Age = 21, StandardID =1 },

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18, StandardID =2 },

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

IList<Standard> standardList = new List<Standard>() {

    new Standard(){ StandardID = 1, StandardName="Standard 1"},

    new Standard(){ StandardID = 2, StandardName="Standard 2"},

    new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var innerJoin = from s in studentList // 外序列

                      join st in standardList //內部序列

                      on s.StandardID equals st.StandardID // 鍵選擇器

                      select new { // 結果選擇器

                                    StudentName = s.StudentName,

                                    StandardName = st.StandardName

                                };

**示例：VB.Net中的查詢語法聯結運運算元**

Dim innerJoin = From s In studentList ' outer sequence

                      Join std In standardList  ' inner sequence

                      On s.StandardID Equals std.StandardID '鍵選擇器

                      Select \_  ' 選擇器結果

                            StudentName = s.StudentName,

                            StandardName = std.StandardName

輸出：

John - Standard 1

Moin - Standard 1

Bill - Standard 2

Ram - Standard 2

使用equals運運算元匹配查詢語法中的鍵選擇器。==無效。

**要記住的要點**

1. Join 和 GroupJoin是連接運運算元。
2. Join 類似於SQL的內部連接。它返回一個新集合，其中包含兩個鍵匹配的集合中的公共元素。
3. **Join** 對內部序列和外部序列這兩個序列進行運算，並生成結果序列。
4. **Join**查詢語法：
5. from... in outerSequence
6. join... in innerSequence
7. on  outerKey equals innerKey

select ...

# LINQ 聯結運運算元 GroupJoin

我們已經在上一節中看到了Join運運算元。GroupJoin運運算元執行與Join運運算元相同的任務，不同之處在於GroupJoin根據指定的組鍵在組中返回結果。GroupJoin運運算元基於鍵聯接兩個序列，並通過匹配鍵將結果分組，然後返回分組的結果和鍵的集合。

GroupJoin需要與Join相同的參數。GroupJoin具有以下兩種重載方法：

**GroupJoin重載方法：**

public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter,

IEnumerable<TInner>, TResult> resultSelector);

public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>,

TResult> resultSelector, IEqualityComparer<TKey> comparer);

正如您在第一個重載中看到的，方法接受五個輸入參數（除了第一個“this”參數）：1）outer 2）inner 3）outerKeySelector 4）inner keyselector 5）resultSelector。請注意，resultSelector是Func委託類型，它的第二個輸入參數是內部序列的IEnumerable類型。

現在，讓我們使用以下Student和Standard類來瞭解GroupJoin，其中Student類包括與Standard類的StandardID匹配的StandardID。

**示例類**

public class Student{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int StandardID { get; set; }

}

public class Standard{

    public int StandardID { get; set; }

    public string StandardName { get; set; }

}

考慮下麵的GroupJoin查詢示例。

**示例：方法語法C＃中的GroupJoin**

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", StandardID =1 },

    new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },

    new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },

    new Student() { StudentID = 4, StudentName = "Ram",  StandardID =2 },

    new Student() { StudentID = 5, StudentName = "Ron" }

};

IList<Standard> standardList = new List<Standard>() {

    new Standard(){ StandardID = 1, StandardName="Standard 1"},

    new Standard(){ StandardID = 2, StandardName="Standard 2"},

    new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var groupJoin = standardList.GroupJoin(studentList,  //內部序列

                                std => std.StandardID, //outerKeySelector

                                s => s.StandardID,     //innerKeySelector

                                (std, studentsGroup) => new // resultSelector

                                {

                                    Students = studentsGroup,

                                    StandarFulldName = std.StandardName

                                });

foreach (var item in groupJoin)

{

    Console.WriteLine(item.StandarFulldName );

    foreach(var stud in item.Students)

        Console.WriteLine(stud.StudentName);

}

輸出：

Standard 1:

John,

Moin,

Standard 2:

Bill,

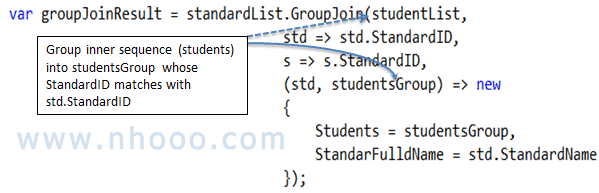
Ram,

Standard 3:

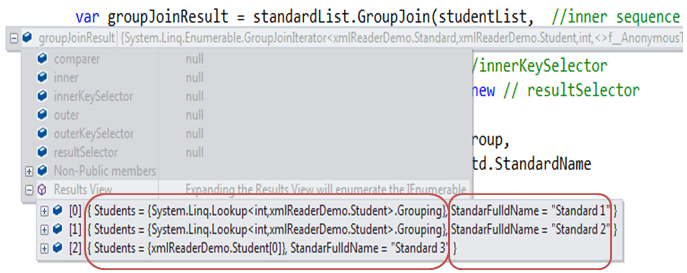
在上面的GroupJoin查詢示例中，standardList是外部序列，因為查詢是從外部序列開始的。GroupJoin方法中的第一個參數是指定內部序列，在上面的示例中為studentList。該方法的第二和第三個參數GroupJoin()是指定一個欄位，該欄位的值應使用lambda運算式進行匹配，以便在結果中包含element。外部序列的鍵選擇器standard => standard.StandardID指示standardList中每個元素的StandardID欄位元應與內部序列studentList的鍵匹配student => student.StandardID。如果兩個鍵欄位的值都匹配，則將該元素包括到分組集合studentsGroup中，其中鍵為StandardID。

Join方法中的最後一個參數是用於表達結果的運算式。在上面的示例中，結果選擇器包括分組的集合studentGroup和StandardName。

下圖說明瞭將內部序列分組到studentsGroup集合中以匹配StandardID鍵，並且可以使用分組的集合來表示結果。

分組運運算元-GroupJoin

結果集將包含具有Students和StandardFullName屬性的匿名物件。學生屬性將是其StandardID與Standard.StandardID匹配的Student的集合。

調試視圖中的GroupJoin結果

您可以使用“ foreach”迴圈訪問結果。每個元素將具有StandardFullName＆Students屬性，其中Student將是一個集合。

**示例：在C＃中訪問GroupJoin結果**

foreach (var item in groupJoinResult)

{

    Console.WriteLine(item.StandarFulldName );

    foreach(var stud in item.Students)

        Console.WriteLine(stud.StudentName);

}

以下是VB.Net中的GroupJoin示例：

**示例：方法VB.Net中的GroupJoin**

Dim groupJoin = standardList.GroupJoin(  ' outer sequence

                    studentList, ' inner sequence

                    Function(s) s.StandardID, ' outerKeySelector

                    Function(stud) stud.StandardID, ' innerKeySelector

                    Function(s, studentGroup) New With { ' result selector

                            .students = studentGroup,

                            .standardName = s.StandardName

                    })

For Each item In groupJoin

    Console.WriteLine(item.standardName)

    For Each std In item.students

            Console.WriteLine( std.StudentName)

    Next

Next

輸出：

Standard 1:

John,

Moin,

Standard 2:

Bill,

Ram,

Standard 3:

## 查詢語法中的GroupJoin

查詢語法中的GroupJoin運運算元與方法語法稍有不同。它需要一個外部序列，內部序列，鍵選擇器和結果選擇器。“ on”關鍵字用於鍵選擇器，其中“等於”運運算元的左側是outerKeySelector，而“等於”運運算元的右側是innerKeySelector。使用**into**關鍵字創建分組的集合。

**語法：查詢語法中的GroupJoin**

from ... in outerSequence

join ... in innerSequence

on outerKey equals innerKey

into groupedCollection

select ...

下面的示例演示了查詢語法中的GroupJoin。

**示例：C＃查詢語法 GroupJoin**

IList<Student> studentList = new List<Student>() {

                    new Student() { StudentID = 1, StudentName = "John", Age = 13, StandardID =1 },

                    new Student() { StudentID = 2, StudentName = "Moin",  Age = 21, StandardID =1 },

                    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18, StandardID =2 },

                    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },

                    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

IList<Standard> standardList = new List<Standard>() {

                    new Standard(){ StandardID = 1, StandardName="Standard 1"},

                    new Standard(){ StandardID = 2, StandardName="Standard 2"},

                    new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var groupJoin = from std in standardList

                    join s in studentList

                    on std.StandardID equals s.StandardID

                    into studentGroup

                    select new {

                              Students = studentGroup ,

                              StandardName = std.StandardName

                          };

foreach (var item in groupJoin)

{

                    Console.WriteLine(item.StandarFulldName );

                    foreach(var stud in item.Students)

                    Console.WriteLine(stud.StudentName);

}

**示例：VB.Net 查詢語法 GroupJoin**

Dim groupJoin = From s In standardList

                Group Join stud In studentList

                On stud.StandardID Equals s.StandardID

                Into Group \_

                Select \_

                StudentsGroup = Group,

                StandardName = s.StandardName

輸出：

Standard 1:

John,

Moin,

Standard 2:

Bill,

Ram,

Standard 3:

在VB.Net版，InTo關鍵字將創建一個具有相同標準的所有學生的組，並將其分配給group關鍵字。所以，在投影結果中使用Group。

**注意：**使用**equals**運運算元來匹配鍵選擇器。==無效。

# LINQ 投影運運算元 Select，SelectMany

LINQ中有兩個可用的投影運運算元。1）Select  2）SelectMany

## Select

Select運運算元始終返回IEnumerable集合，該集合包含基於轉換函數的元素。它類似於產生平面結果集的SQL的Select子句。

現在，讓我們瞭解使用以下Student類的Select查詢運運算元。

public class Student{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

## 在查詢語法中的Select

LINQ查詢語法必須以**Select** 或**GroupBy**子句結尾。下麵的示例演示了Select 運運算元，該運運算元返回StudentName的字串集合。

示例：在查詢語法C＃中Select

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John" },

    new Student() { StudentID = 2, StudentName = "Moin" },

    new Student() { StudentID = 3, StudentName = "Bill" },

    new Student() { StudentID = 4, StudentName = "Ram" },

    new Student() { StudentID = 5, StudentName = "Ron" }

};

var selectResult = from s in studentList

                   select s.StudentName;

選擇運運算元可用於根據我們的要求制定結果。它可用於返回自訂類或匿名類型的集合，其中包括根據我們的需要的屬性。

下麵的select子句示例返回一個包含Name和Age屬性的匿名類型的集合。

示例：在查詢語法C＃中Select

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,

    new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

// 返回具有Name和Age屬性的匿名物件的集合

var selectResult = from s in studentList

                   select new { Name = "Mr. " + s.StudentName, Age = s.Age };

// 反覆運算selectResult

foreach (var item in selectResult)

    Console.WriteLine("Student Name: {0}, Age: {1}", item.Name, item.Age);

示例：在查詢語法VB.Net中選擇

Dim selectResult = From s In studentList

                   Select New With {.Name = s.StudentName, .Age = s.Age}

輸出：

Student Name: Mr. John, Age: 13

Student Name: Mr. Moin, Age: 21

Student Name: Mr. Bill, Age: 18

Student Name: Mr. Ram, Age: 20

Student Name: Mr. Ron, Age: 15

## 在方法語法中Select

Select運運算元在方法語法中是可選的。但是，您可以使用它來塑造資料。在以下示例中，Select擴展方法返回具有Name和Age屬性的匿名物件的集合：

**示例：C＃在方法語法中的Select**

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

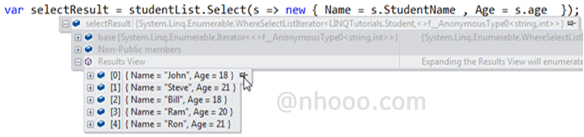
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

var selectResult = studentList.Select(s => new { Name = s.StudentName ,

                                                 Age = s.Age  });

在上面的示例中，selectResult將包含具有Name和Age屬性的匿名物件，如下麵的調試視圖所示。

Select子句返回一個匿名物件

示例：VB.Net中在方法語法Select

Dim selectResult = studentList.Select(Function(s) New With {.Name = s.StudentName,

                                                            .Age = s.Age})

## SelectMany

SelectMany 運運算元投射基於轉換函數的值序列，然後將它們扁平化為一個序列。

# LINQ 限定運運算元

限定運運算元在某些條件下評估序列的元素，然後返回布林值以指示某些或所有元素都滿足條件。

| **運運算元** | **描述** |
| --- | --- |
| All | 檢查序列中的所有元素是否滿足指定的條件 |
| Any | 檢查序列中是否有任一元素滿足指定條件 |
| Contains | 檢查序列是否包含特定元素 |

## All

All運運算元在指定條件下評估給定集合中的每個元素，如果所有元素均滿足條件，則返回True。

示例：C# All運運算元

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

// 檢查所有學生是否都是青少年

bool areAllStudentsTeenAger = studentList.All(s => s.Age > 12 && s.Age < 20);

Console.WriteLine(areAllStudentsTeenAger);

示例：VB.Net All 運運算元

Dim areAllStudentsTeenAger = studentList.All(Function(s) s.Age > 12 And s.Age < 20)

輸出：

false

## Any

Any檢查元素是否滿足給定條件？在以下示例中，“Any”操作用於檢查是否有任何學生是青少年。

**示例：Any 運運算元 與 C＃**

bool isAnyStudentTeenAger = studentList.Any(s => s.age > 12 && s.age < 20);

**示例：Any運運算元 與 VB.Net**

Dim isAnyStudentTeenAger = studentList.Any(Function(s) s.Age > 12 And s.Age < 20)

輸出：

true

c#查詢語法不支援限定運運算元。

## Contains

Contains方法用來確定序列是否包含滿足指定條件的元素。如果有返回true，否則返回false。以下代碼使用默認的String比較器來判斷序列中是否含有指定的字串：

string[] source1 = new string[] { "A", "B", "C", "D", "E", "F" };

Console.WriteLine(source1.Contains("A")); //輸出 "True"

Console.WriteLine(source1.Contains("G")); //輸出  "False"

瞭解有關限定運運算元的資訊- Contains 在下一部分中。

**LINQ 限定運運算元 Contains**

Contains運運算元檢查集合中是否存在指定的元素，並返回布林值。

Contains()擴展方法有以下兩個重載。第一個重載方法需要一個值來檢入集合，第二個重載方法需要使用IEqualityComparer類型的附加參數來進行自訂相等性比較。

**Contains() 重載：**

public static bool Contains<TSource>(this IEnumerable<TSource> source, TSource value);

public static bool Contains<TSource>(this IEnumerable<TSource> source,

                                     TSource value,

                                    IEqualityComparer<TSource> comparer);

如上所述，Contains()擴展方法需要一個值作為輸入參數進行檢查。值的類型必須與泛型集合的類型相同。下面的示例包含檢查集合中是否存在10。請注意，int是泛型集合的一種類型。

**示例：C# 中 Contains運運算元**

IList<int> intList = new List<int>() { 1, 2, 3, 4, 5 };

bool result = intList.Contains(10);  // 返回 false

**示例：VB.Net Contains 運運算元**

Dim intList As IList(Of Integer) = New List(Of Integer) From {1, 2, 3, 4, 5}

Dim result = intList.Contains(10)  '返回 false

上面的示例適用於原始資料類型。但是，它不適用於自訂類。看以下示例：

Error：return false

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

Student std = new Student(){ StudentID =3, StudentName = "Bill"};

bool result = studentList.Contains(std); //returns false

正如您在上面的示例中看到的，Contains返回false，即使studentList中存在“Bill”。這是因為Contains擴展方法只比較物件的引用，而不比較物件的實際值。所以要比較student物件的值，需要通過實現IEqualityComparer介面創建一個類，該介面比較兩個student物件的值並返回boolean。

以下是StudentComparer類，它實現IEqualityComparer<Student>介面來比較兩個Students物件的值：

示例：IEqualityComperer

class StudentComparer : IEqualityComparer<Student>

{

        public bool Equals(Student x, Student y)

        {

            if (x.StudentID == y.StudentID &&

                        x.StudentName.ToLower() == y.StudentName.ToLower())

                return true;

            return false;

        }

        public int GetHashCode(Student obj)

        {

            return obj.GetHashCode();

        }

}

現在，你可以在Contains擴展方法的第二個重載方法中使用上面的StudentComparer類，該方法接受第二個參數為IEqualityComparer類型，如下所示:

**示例: c # 中 Contains 與 Comparer 類**

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

Student std = new Student(){ StudentID =3, StudentName = "Bill"};

bool result = studentList.Contains(std, new StudentComparer()); //returns true

因此，必須使用Comparer類才能從自訂類的Contains擴展方法中獲得正確的結果。

下麵是 VB.Net 中的一個類似的實例:

示例： VB.Net中 Comparer 類的  Contains

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

Public Class StudentComparer Implements IEqualityComparer(Of Student)

    Public Function Equals1(x As Student, y As Student) As Boolean Implements IEqualityComparer(Of Student).Equals

        If (x.StudentID = y.StudentID And x.StudentName.ToLower() = y.StudentName.ToLower()) Then

            Return True

        End If

        Return False

    End Function

    Public Function GetHashCode1(obj As Student) As Integer Implements IEqualityComparer(Of Student).GetHashCode

        Return obj.GetHashCode()

    End Function

End Class

Sub Main

    Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 18},

        New Student() With {.StudentID = 2, .StudentName = "Steve", .Age = 15},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 25},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 19}

    }

    Dim std As New Student With {.StudentID = 3, .StudentName = "Bill"}

    Dim result = studentList.Contains(std, New StudentComparer()) ' returns true

End Sub

在C#或VB.Net中，不支持限定詞運運算元。

**注意事項**

1. All, Any & Contains是LINQ中的限定運運算元。
2. All檢查序列中的所有元素是否滿足指定條件。
3. 檢查序列中是否有任何元素滿足指定條件
4. Contains 操作符檢查指定的元素是否存在于集合中。
5. 使用派生 IEqualityOperator 和 Contains 的自訂類檢查集合中的物件。
6. 在C#或VB.Net的查詢語法中不支援All, Any & Contains。

下一步瞭解聚合運運算元。

# LINQ 聚合運運算元 Aggregate

聚合運運算元對集合中元素的數值屬性執行數學運算，如Average、Aggregate、Count、Max、Min和Sum。

| **方法** | **描述** |
| --- | --- |
| Aggregate | 對集合中的值執行自訂聚合操作。 |
| Average | 計算集合中數位元項的平均值。 |
| Count | 統計集合中的元素。 |
| LongCount | 統計集合中的元素。 |
| Max | 查找集合中的最大值。 |
| Min | 查找集合中的最小值。 |
| Sum | 計算集合中值的總和。 |

## Aggregate

聚合方法執行累加操作。聚合擴展方法具有以下重載方法：

Aggregate()重載：

public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,

                                         Func<TSource, TSource, TSource> func);

public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,

                                         TAccumulate seed,

                                         Func<TAccumulate, TSource, TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<TSource> source,

                                         TAccumulate seed,

                                         Func<TAccumulate, TSource, TAccumulate> func,

                                         Func<TAccumulate, TResult> resultSelector);

下面的示例演示了 Aggregate 方法，該方法返回字串清單中逗號分隔的元素。

示例：C＃方法語法的中Aggregate

IList<String> strList = new List<String>() { "One", "Two", "Three", "Four", "Five"};

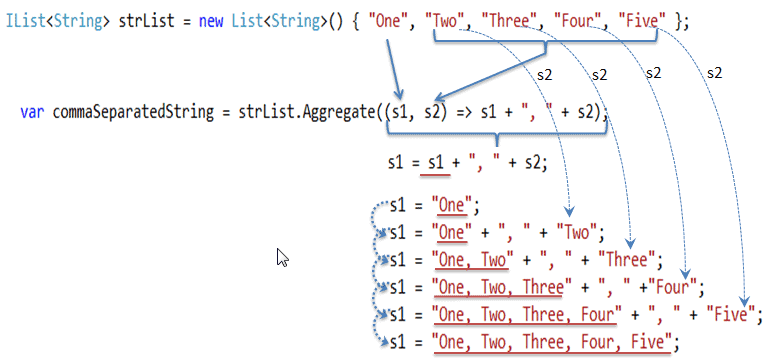
var commaSeperatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);

Console.WriteLine(commaSeperatedString);

輸出：

 One, Two, Three, Four, Five

在上面的示例中，Aggregate擴展方法從strList集合返回逗號分隔的字串。下圖說明瞭以上示例中執行的整個聚合操作。

聚合擴展方法

如上圖所示，strList“ One”的第一項將作為 s1傳遞，其餘項將作為 s2傳遞。Lambda 運算式(s1，s2) = > s1 + ","+ s2將被視為 s1 = s1 +","+ s1，其中 s1將為集合中的每個項累積。因此，Aggregate 方法將返回逗號分隔的字串。

示例：VB.Net中的方法語法 的Aggregate

Dim strList As IList(Of String) = New List(Of String) From {

                                                            "One",

                                                            "Two",

                                                            "Three",

                                                            "Four",

                                                            "Five"

                                                        }

Dim commaSeparatedString = strList.Aggregate(Function(s1, s2) s1 + ", " + s2)

## 帶種子值的聚合方法

Aggregate的第二個重載方法需要第一個參數來累積種子值。第二個參數是Func類型的委託：

TAccumulate Aggregate<TSource, TAccumulate>(TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func);

下面的示例在Aggregate擴展方法中將字串用作種子值。

示例：C＃ 帶有種子值的聚合

// 學生集合

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

string commaSeparatedStudentNames = studentList.Aggregate<Student, string>(

                                        "Student Names: ",  // 種子價值

                                        (str, s) => str += s.StudentName + "," );

Console.WriteLine(commaSeparatedStudentNames);

示例：VB.Net 具有種子值的聚合

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

    }

 Dim commaSeparatedStudentNames = studentList.Aggregate(Of String)(

               "Student Names: ",

               Function(str, s) str + s.StudentName + ",")

Console.WriteLine(commaSeparatedStudentNames);

輸出：

Student Names: John, Moin, Bill, Ram, Ron,

在上面的示例中，Aggregate 方法的第一個參數是“ Student Names: ”字串，該字串將與所有學生名一起累積。Lambda 運算式中的逗號將作為第二個參數傳遞。

下面的示例使用 Aggregate 運運算元添加所有學生的年齡。

示例：帶有種子值C＃的聚合

// 學生集合

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

int SumOfStudentsAge = studentList.Aggregate<Student, int>(0,

                                                (totalAge, s) => totalAge += s.Age  );

## 帶有結果選擇器的聚合方法

現在，讓我們看看第三個重載方法，它需要 Func 委託運算式的第三個參數作為結果選擇器，這樣您就可以公式化結果。

示例：C＃使用結果選擇器進行聚合

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

string commaSeparatedStudentNames = studentList.Aggregate<Student, string,string>(

                                            String.Empty, // 種子值

                                            (str, s) => str += s.StudentName + ",", // 使用種子值返回結果，String.Empty以str的形式進入lambda運算式

                                            str => str.Substring(0,str.Length - 1 )); // 刪除最後一個逗號的結果選擇器

Console.WriteLine(commaSeparatedStudentNames);

在上面的示例中，我們指定了一個lambda運算式str => str.Substring(0,str.Length - 1 )，該運算式將刪除字串結果中的最後一個逗號。下麵是VB.Net中的相同示例。

示例：VB.Net結果選擇器與聚合

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

    }

Dim commaSeparatedStudentNames = studentList.Aggregate(Of String, String)(

               String.Empty,

               Function(str, s) str + s.StudentName + ",",

               Function(str) str.Substring(0, str.Length - 1))

Console.WriteLine(commaSeparatedStudentNames);

輸出：

John, Moin, Bill, Ram, Ron

C # 或 VB.Net 中的查詢語法不支援聚合運運算元。

在下一部分中瞭解另一種合計運運算元-Average （計算平均值）。

# LINQ 聚合運運算元 Average

Average 擴展方法計算集合中數值項的平均值。Average 方法返回可空或不可空的十進位值、雙值或浮點值。

下面的示例演示Agerage方法，該方法返回集合中所有整數的平均值。

示例：平均方法C＃

IList<int> intList = new List<int>>() { 10, 20, 30 };

var avg = intList.Average();

Console.WriteLine("Average: {0}", avg);

您可以將類的 int、 decimal、 double 或 float 屬性指定為 lambda 運算式，希望獲得其平均值。下面的示例演示複雜類型上的 Average 方法。

示例：C＃中的方法語法Average

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

var avgAge = studentList.Average(s => s.Age);

Console.WriteLine("Average Age of Student: {0}", avgAge);

示例：VB.Net中 方法語法的Average

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim avgAge = studentList.Average(Function(s) s.Age)

Console.WriteLine("學生的平均年齡: {0}", avgAge)

輸出：

學生的平均年齡：17.4

C # 不支援查詢語法中的 Average 運運算元。但是，VB.Net 支援它，如下所示。

示例：VB.Net 查詢語法中的Average

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim avgAge = Aggregate st In studentList Into Average(st.Age)

Console.WriteLine("學生的平均年齡: {0}", avgAge)

輸出：

學生的平均年齡：17.4

瞭解另一個聚合運運算元-下一節中的Count。

# LINQ 聚合運運算元 Count

Count運運算元返回集合中的元素數或滿足給定條件的元素數。

Count()擴展方法有以下兩種重載：

**Count() 重載：**

int Count<TSource>();int Count<TSource>(Func<TSource, bool> predicate);

Count的第一個重載方法返回指定集合中的元素數，而第二個重載方法返回滿足指定條件的元素數（由lambda運算式/predicate 謂詞函數給出）。

下麵的示例演示了原語集合上的Count()。

示例：Count()與C＃

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50 };

var totalElements = intList.Count();

Console.WriteLine("元素總數: {0}", totalElements);

var evenElements = intList.Count(i => i%2 == 0);

Console.WriteLine("偶數元素: {0}", evenElements);

輸出：

元素總數：5

偶數元素：3

下面的示例演示Count()複雜類型集合上的方法。

示例：在C＃中的Count()

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Mathew" , Age = 15 }

    };

var totalStudents = studentList.Count();

Console.WriteLine("學生總人數: {0}", totalStudents);

var adultStudents = studentList.Count(s => s.Age >= 18);

Console.WriteLine("成人學生人數: {0}", adultStudents );

輸出：

學生總人數：5

成人學生人數：3

示例：在VB.Net中Count()

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim totalStudents = studentList.Count()

Console.WriteLine("學生總人數: {0}", totalStudents)

輸出：

學生總人數：5

在VB.Net中不支援帶有謂詞參數的Count(predicate)擴展方法。

## 查詢語法中的Count運運算元

示例：查詢語法中的Count運運算元-VB.Net

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim totalStudents = Aggregate st In studentList

                    Into Count(st.Age >= 18)

Console.WriteLine("學生總人數: {0}", totalStudents)

輸出：

 學生總人數：3

C＃查詢語法不支援聚合運運算元。但是，您可以將查詢括在方括號中並使用彙總函式，如下所示。

示例：查詢語法C＃中的Count運運算元

var totalAge = (from s in studentList

                select s.age).Count();

在下一部分中瞭解另一個聚合運運算元-Max。

# LINQ 聚合運運算元 Max

Max()方法返回集合中最大的數值元素。

下面的示例演示原始集合上的Max()方法。

示例：C＃中的Max()方法

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var largest = intList.Max();

Console.WriteLine("最大元素: {0}", largest);

var largestEvenElements = intList.Max(i => {

                        if(i%2 == 0)

                        return i;

                        return 0;

                        });

Console.WriteLine("最大偶數: {0}", largestEvenElements );

輸出：

最大元素：87

最大偶數：50

下面的示例演示複雜類型集合上的Max()方法。

示例：C＃方法語法中的Max()方法

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

var oldest = studentList.Max(s => s.Age);

Console.WriteLine("Oldest Student Age: {0}", oldest);

示例：VB.Net方法語法中的Max方法

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim oldest = studentList.Max(Function(s) s.Age)

Console.WriteLine("最大的學生年齡: {0}", oldest)

輸出：

最大的學生年齡：21

Max返回任何資料類型的結果。以下示例顯示了如何找到集合中 名稱最長 的學生：

示例：在C＃中Max()

public class Student : IComparable<Student>

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

    public int StandardID { get; set; }

    public int CompareTo(Student other)

    {

        if (this.StudentName.Length >= other.StudentName.Length)

            return 1;

        return 0;

    }

}

class Program

{

    static void Main(string[] args)

    {

        // 學生集合

        IList<Student> studentList = new List<Student>>() {

                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

                new Student() { StudentID = 5, StudentName = "Steve" , Age = 15 }

            };

        var studentWithLongName = studentList.Max();

        Console.WriteLine("Student ID: {0}, Student Name: {1}",

                                        .StudentID, studentWithLongName.StudentName)

    }

}

輸出：

Student ID：5，StudentName ：Steve

您可以使用與Max相同的方式使用Min擴展方法/運運算元。

根據上面的實例，要找到名字最長的學生，需要實現IComparable<T>介面，並在CompareTo方法中比較學生名字的長度。現在，您可以使用Max()方法，它將使用CompareTo方法來返回適當的結果。

## 查詢語法中的Max運運算元

C#查詢語法不支援Max運運算元。但是，它在VB.Net版查詢語法如下所示。

示例：查詢語法中的Max運運算元 - VB.Net

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim maxAge = Aggregate st In studentList Into Max(st.Age)

Console.WriteLine("最大學生年齡: {0}", maxAge);

輸出：

最大學生年齡：21

在下一部分中瞭解另一個聚合運運算元-Sum。

# LINQ 聚合運運算元 Sum

Sum()方法計算集合中數位元項的總和。

下面的示例演示原始集合上的Sum()方法。

示例：LINQ Sum() - C＃

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var total = intList.Sum();

Console.WriteLine("總計: {0}", total);

var sumOfEvenElements = intList.Sum(i => {

                    if(i%2 == 0)

                    return i;

                    return 0;

                        });

Console.WriteLine("偶數元素的總計: {0}", sumOfEvenElements );

輸出：

總計：243

偶數元素的總計：90

下面的示例計算學生集合中所有學生的年齡總和，以及成年學生的人數。

示例：LINQ Sum() 計算學生年齡總和和成年學生人數 - C＃

IList<Student> studentList = new List<Student>>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

        new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

    };

var sumOfAge = studentList.Sum(s => s.Age);

Console.WriteLine("學生年齡總和: {0}", sumOfAge);

var numOfAdults = studentList.Sum(s => {

if(s.Age >= 18)

    return 1;

else

    return 0;

});

Console.WriteLine("成年學生人數: {0}", numOfAdults);

示例：VB.NET中方法語法的Sum

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim sumOfAge = studentList.Sum(Function(s) s.Age)

Console.WriteLine("學生年齡總和: {0}", sumOfAge)

Dim numOfAdults = studentList.Sum(Function(s)

            if(s.Age >= 18)

                return 1

            else

                return 0

            end if

        end function)

Console.WriteLine("成年學生人數: {0}", numOfAdults)

輸出：

學生年齡總和：87

成年學生人數：3

## 查詢語法中的Sum運運算元

C# 查詢語法中不支援Sum運運算元。

示例：查詢語法中的Sum運運算元 - VB.Net

// 學生集合

Dim studentList = New List(Of Student) From {

        New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

        New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

        New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

        New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

        New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

Dim totalAge = Aggregate st In studentList Into Sum(st.Age)

Console.WriteLine("所有年齡段的總和: {0}", totalAge);

輸出：

所有年齡段的總和：87

# LINQ 元素運運算元 ElementAt，ElementAtOrDefault

元素運運算元從序列（集合）中返回特定元素。

下表列出了LINQ中的所有Element運運算元。

| **元素運運算元（方法）** | **描述** |
| --- | --- |
| ElementAt | 返回集合中指定索引處的元素 |
| ElementAtOrDefault | 返回集合中指定索引處的元素；如果索引超出範圍，則返回預設值。 |
| First | 返回集合的第一個元素，或滿足條件的第一個元素。 |
| FirstOrDefault | 返回集合的第一個元素，或滿足條件的第一個元素。如果索引超出範圍，則返回預設值。 |
| Last | 返回集合的最後一個元素，或滿足條件的最後一個元素 |
| LastOrDefault | 返回集合的最後一個元素，或滿足條件的最後一個元素。如果不存在這樣的元素，則返回預設值。 |
| Single | 返回集合中的唯一元素，或唯一滿足條件的元素。 |
| SingleOrDefault | 返回集合中的唯一元素，或唯一滿足條件的元素。如果不存在這樣的元素，或者該集合不完全包含一個元素，則返回預設值。 |

ElementAt()方法從給定集合返回指定索引中的元素。如果指定的索引超出集合的範圍，則它將拋出“索引超出範圍（Index out of range exception）”異常。請注意，索引是從零開始的索引。

ElementAtOrDefault()方法還從協作中返回指定索引中的元素，如果指定索引不在集合的範圍內，則它將返回資料類型的預設值，而不是引發錯誤。

下面的示例演示原始集合上的ElementAt和ElementAtOrDefault方法。

示例：LINQElementAt()和ElementAtOrDefault()方法 - C＃

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { "One", "Two", null, "Four", "Five" };

Console.WriteLine("intList中的第一個元素: {0}", intList.ElementAt(0));

Console.WriteLine("strList中的第一個元素: {0}", strList.ElementAt(0));

Console.WriteLine("intList中的第二個元素: {0}", intList.ElementAt(1));

Console.WriteLine("strList中的第二個元素: {0}", strList.ElementAt(1));

Console.WriteLine("intList中的第三個元素: {0}", intList.ElementAtOrDefault(2));

Console.WriteLine("strList中的第三個元素: {0}", strList.ElementAtOrDefault(2));

Console.WriteLine("intList中的第10個元素: {0} - 默認int值",

                intList.ElementAtOrDefault(9));

Console.WriteLine("strList中的第十個元素: {0} - 預設字串值(null)",

                 strList.ElementAtOrDefault(9));

Console.WriteLine("intList. ElementAt (9)拋出異常: 索引超出範圍");

Console.WriteLine("-------------------------------------------------------------");

Console.WriteLine(intList.ElementAt(9));

輸出：

intList中的第一個元素：10

strList中的第一個元素：

intList中的第二元素：21

strList中的第二個元素：

intList中的第三個元素：30

strList中的第三個元素：

intList中的第十個元素：0 - 默認int值

strList中的第十個元素：- 預設字串值(null)

----------------------------- --------------------------------

intList. ElementAt (9)拋出異常: 索引超出範圍

正如您在上面示例中所看到的那樣，intList.ElementAtOrDefault(9)返回0(預設值為int)，因為intList不包含第十元素。 然而，intList.ElementAt(9)以相同的方式拋出“索引超出範圍”，(9)返回null，它是字串類型的預設值。 (控制台顯示空白空間，因為它不能顯示空)

因此，建議使用 ElementAtOrDefault 擴展方法來消除運行時異常的可能性。

在下一節中，將瞭解另一個元素運運算元First和FirstOrDefault。

# LINQ 元素運運算元 First和FirstOrDefault

First和FirstOrDefault方法從集合中第零個索引返回一個元素，即第一個元素。另外，它返回滿足指定條件的元素。

| **元素運運算元** | **描述** |
| --- | --- |
| First | 返回集合的第一個元素，或滿足條件的第一個元素。 |
| FirstOrDefault | 返回集合的第一個元素，或滿足條件的第一個元素。如果索引超出範圍，則返回預設值。 |

First和FirstOrDefault具有兩個重載方法。第一個重載方法不使用任何輸入參數，並返回集合中的第一個元素。第二個重載方法將lambda運算式作為謂詞委託來指定條件，然後返回滿足指定條件的第一個元素。

First() 和FirstOrDefault()重載：

public static TSource First<TSource>(this IEnumerable<TSource> source);

public static TSource First<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

First()方法使用lambda運算式或Func委託返回集合的第一個元素，或滿足指定條件的第一個元素。如果給定的集合為空或不包含任何滿足條件的元素，則它將拋出 InvalidOperation 異常。

FirstOrDefault()方法與First()方法具有相同的作用。唯一的區別是，如果集合為空或找不到滿足條件的任何元素，它將返回集合資料類型的預設值。

下面的示例演示First()方法。

示例：LINQ First()方法 - C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("intList中的第一個元素: {0}", intList.First());

Console.WriteLine("intList中的第一個偶數元素: {0}", intList.First(i => i % 2 == 0));

Console.WriteLine("strList中的第一個元素： {0}", strList.First());

Console.WriteLine("emptyList.First()拋出InvalidOperationException");

Console.WriteLine("-------------------------------------------------------------");

Console.WriteLine(emptyList.First());

輸出：

intList中的第一個元素：7

intList中的第一個偶數元素：10

strList中的第一個元素：

emptyList.First()拋出InvalidOperationException

----------------------------------------------- --------------

運行時異常：序列不包含任何元素...

下面的示例演示FirstOrDefault()方法。

示例：LINQ FirstOrDefault() - C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("intList中的第一個元素: {0}", intList.FirstOrDefault());

Console.WriteLine("intList中的第一個偶數元素: {0}",intList.FirstOrDefault(i => i % 2 == 0));

Console.WriteLine("strList中的第一個元素: {0}", strList.FirstOrDefault());

Console.WriteLine("emptyList中的第一個元素: {0}", emptyList.FirstOrDefault());

輸出：

intList中的第一個元素：7

intList中的第一個偶數元素：10

strList中的第一個元素：

emptyList中的第一個元素：

在First()或FirstOrDefault()中指定條件時要小心。如果集合不包含任何滿足指定條件的元素或包含null元素，則First()將拋出異常。

如果集合包含空元素，則 FirstOrDefault ()在計算指定條件時拋出異常。下麵的示例演示了這一點。

示例：LINQFirst()和FirstOrDefault() - C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("intList中大於250的第一個元素: {0}", intList.First( i > 250));

Console.WriteLine("在 intList 中的第一個偶數元素: {0}", strList.FirstOrDefault(s => s.Contains("T")));

輸出：

 Run-time exception: Sequence contains no matching element

//運行時異常：序列不包含匹配元素

# LINQ 元素運運算元 Last和LastOrDefault

| **元素運運算元** | **描述** |
| --- | --- |
| Last | 返回集合中的最後一個元素，或滿足條件的最後一個元素 |
| LastOrDefault | 返回集合中的最後一個元素，或滿足條件的最後一個元素。如果不存在這樣的元素，則返回預設值。 |

Last和LastOrDefault具有兩個重載方法。一個重載方法不使用任何輸入參數，而是返回集合中的最後一個元素。第二個重載方法使用lambda運算式指定條件，然後返回滿足指定條件的最後一個元素。

Last() 和 LastOrDefault()重載：

public static TSource Last<TSource>(this IEnumerable<TSource> source);

public static TSource Last<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

Last()方法從集合中返回最後一個元素，或者使用lambda運算式或Func委託返回滿足指定條件的最後一個元素。如果給定的集合為空或不包含任何滿足條件的元素，則它將拋出 InvalidOperation 異常。

LastOrDefault()方法與Last()方法具有相同的作用。唯一的區別是，如果集合為空或找不到滿足條件的任何元素，它將返回集合資料類型的預設值。

下面的示例演示Last()方法。

示例：LINQ Last() - C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("intList中的最後一個元素: {0}", intList.Last());

Console.WriteLine("intList中的最後一個偶數: {0}", intList.Last(i => i % 2 == 0));

Console.WriteLine("strList中的最後一個元素: {0}", strList.Last());

Console.WriteLine("emptyList.Last()拋出InvalidOperationException ");

Console.WriteLine("-------------------------------------------------------------");

Console.WriteLine(emptyList.Last());

輸出：

intList中的最後一個元素：

intList中的最後一個偶數：50

strList中的最後一個元素：5

emptyList.Last()拋出InvalidOperationException

----------------------------------------------- --------------

Run-time exception: Sequence contains no elements...

下面的示例演示LastOrDefault()方法。

示例：LINQ LastOrDefault()-C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("intList中的最後一個元素: {0}", intList.LastOrDefault());

Console.WriteLine("intList中的最後一個偶數元素: {0}",intList.LastOrDefault(i => i % 2 == 0));

Console.WriteLine("strList中的最後一個元素: {0}", strList.LastOrDefault());

Console.WriteLine("emptyList中的最後一個元素: {0}", emptyList.LastOrDefault());

輸出：

intList中的最後一個元素：

intList中的最後一個偶數元素：10

strList中的最後一個元素：

emptyList中的最後一個元素：

在Last()或LastOrDefault()中指定條件時要小心。如果集合不包含任何滿足指定條件的元素或包含null元素，那麼Last()將拋出異常。

如果集合包含null元素，則LastOrDefault()在評估指定條件時會引發異常。以下示例對此進行了演示。

示例：LINQLast()和LastOrDefault() - C＃

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("intList中大於250的最後一個元素: {0}", intList.Last(i => i > 250));

Console.WriteLine("intList中的最後一個偶數元素: {0}", strList.LastOrDefault(s => s.Contains("T")));

輸出：

Run-time exception: Sequence contains no matching element

//運行時異常：序列不包含匹配元素

**LINQ 元素運運算元 Single和SingleOrDefault**

| **元素運運算元** | **描述** |
| --- | --- |
| Single | 返回集合中的唯一元素，或唯一滿足條件的元素。如果Single()在集合中未找到任何元素或一個以上的元素，則拋出InvalidOperationException。 |
| SingleOrDefault | 與Single相同，不同之處在於它返回指定泛型類型的預設值，而不是在找不到指定條件的元素時拋出異常。但是，如果在集合中為指定條件找到了多個元素，它將拋出InvalidOperationException。 |

Single和SingleOrDefault具有兩個重載方法。第一個重載方法不使用任何輸入參數，並返回集合中的單個元素。第二種重載方法將lambda運算式作為指定條件的謂詞委託，並返回滿足指定條件的單個元素。

Single()和SingleOrDefault()重載：

public static TSource Single<TSource>(this IEnumerable<TSource> source);

public static TSource Single<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

Single()返回集合中的唯一元素，或唯一滿足指定條件的元素。如果給定的集合不包含任何元素或包含多個元素，則Single()拋出InvalidOperationException。

SingleOrDefault()方法與Single()方法具有相同的作用。唯一的區別是，如果集合為空，不包含一個以上元素或對於指定條件不包含一個或多個元素，則它返回集合資料類型的預設值。

示例：Single方法語法 - C＃

IList<int> oneElementList = new List<int>() { 7 };

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("oneElementList 中的唯一元素: {0}", oneElementList.Single());

Console.WriteLine("oneElementList 中的唯一元素: {0}",oneElementList.SingleOrDefault());

Console.WriteLine("emptyList中的元素: {0}", emptyList.SingleOrDefault());

Console.WriteLine("intList中唯一小於10的元素: {0}",intList.Single(i => i < 10));

//下麵拋出一個異常

//Console.WriteLine("intList中唯一的元素: {0}", intList.Single());

//Console.WriteLine("intList中唯一的元素: {0}", intList.SingleOrDefault());

//Console.WriteLine("emptyList中唯一的元素: {0}", emptyList.Single());

輸出：

oneElementList中的唯一元素：7

oneElementList中的唯一元素：7

emptyList中的元素：0

intList中唯一小於10的元素：7

以下示例代碼將引發異常，因為Single()或SingleOrDefault()對於指定條件不返回任何元素或返回多個元素。

C＃：Single()和SingleOrDefault()

IList<int> oneElementList = new List<int>() { 7 };

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

//下麵拋出錯誤，因為清單包含多個小於100的元素

Console.WriteLine("intList中小於100的元素: {0}", intList.Single(i => i < 100));

//下麵拋出錯誤，因為清單包含多個小於100的元素

Console.WriteLine("intList中小於100的元素: {0}", intList.SingleOrDefault(i => i < 100));

//由於清單包含多個元素，下麵拋出錯誤

Console.WriteLine("intList中唯一的元素: {0}", intList.Single());

//由於清單包含多個元素，下麵拋出錯誤

Console.WriteLine("intList中唯一的元素: {0}", intList.SingleOrDefault());

//下麵拋出錯誤，因為清單不包含任何元素

Console.WriteLine("emptyList 中的唯一元素: {0}", emptyList.Single());

**要記住的要點**

1. Single()要求集合中只有一個元素。
2. Single() 當集合中沒有任何元素或一個以上元素時，拋出異常。
3. 如果在Single()中指定了一個條件，並且結果不包含任何元素或包含多個元素，則會引發異常。
4. 如果集合或指定條件中沒有元素，SingleOrDefault()將返回泛型集合的資料類型的預設值。
5. 如果集合或指定條件中有多個元素，SingleOrDefault()將引發異常。

**LINQ 相等運運算元 SequenceEqual**

只有一個相等運運算元：SequenceEqual。SequenceEqual方法檢查兩個集合中的元素數量，每個元素的值和元素的順序是否相等。

如果集合包含原始資料類型的元素，則它將比較元素的值和數量，而具有複雜類型元素的集合將檢查物件的引用。因此，如果物件具有相同的引用，則將它們視為相等，否則將其視為不相等。

下面的示例演示了帶有原始資料類型集合的SequenceEqual方法。

示例：方法語法C＃中的SequenceEqual

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

IList<string> strList2 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

bool isEqual = strList1.SequenceEqual(strList2); // 返回true

Console.WriteLine(isEqual);

輸出：

True

如果元素的順序不同，則SequenceEqual()方法返回false。

示例：C＃中方法語法的SequenceEqual

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

IList<string> strList2 = new List<string>(){ "Two", "One", "Three", "Four", "Three"};

bool isEqual = strList1.SequenceEqual(strList2); // 返回false

Console.WriteLine(isEqual);

輸出：

false

SequenceEqual擴展方法檢查兩個物件的引用，以確定兩個序列是否相等。這可能會給出錯誤的結果。看以下示例：

示例：C＃中的SequenceEqual

Student std = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList1 = new List<Student>(){ std };

IList<Student> studentList2 = new List<Student>(){ std };

bool isEqual = studentList1.SequenceEqual(studentList2); // 返回true

Student std1 = new Student() { StudentID = 1, StudentName = "Bill" };

Student std2 = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList3 = new List<Student>(){ std1};

IList<Student> studentList4 = new List<Student>(){ std2 };

isEqual = studentList3.SequenceEqual(studentList4);// 返回false

在上面的示例中，studentList1和studentList2包含相同的學生對象std。因此studentList1.SequenceEqual(studentList2)返回true。但是，stdList1和stdList2包含兩個單獨的學生對象std1和std2。所以現在，即使std1和std2包含相同的值，stdList1.SequenceEqual(stdList2)也將返回false。

要比較兩個複雜類型（參考類型或物件）集合的值，需要實現IEqualityComperar <T>介面，如下所示。

示例：IEqualityComparer C＃：

class StudentComparer : IEqualityComparer<Student>

{

    public bool Equals(Student x, Student y)

    {

        if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

            return true;

        return false;

    }

    public int GetHashCode(Student obj)

    {

        return obj.GetHashCode();

    }

}

現在，您可以使用SequenceEqual擴展方法中的上述StudentComparer類作為第二個參數來比較值：

示例：C＃使用SequenceEqual比較物件類型元素

IList<Student> studentList1 = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

IList<Student> studentList2 = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

// 以下返回true

bool isEqual = studentList1.SequenceEqual(studentList2, new StudentComparer());

**要記住的要點**

1. SequenceEqual方法比較項目數及其原始資料類型的值。
2. SequenceEqual方法比較物件對複雜資料類型的引用。
3. 使用IEqualityComparer類可通過SequenceEqual方法比較兩個複雜類型的集合。

# LINQ 串聯運運算元 Concat

Concat()方法附加兩個相同類型的序列，並返回一個新序列（集合）。

示例：C＃中的Concat

IList<string> collection1 = new List<string>() { "One", "Two", "Three" };

IList<string> collection2 = new List<string>() { "Five", "Six"};

var collection3 = collection1.Concat(collection2);

foreach (string str in collection3)

    Console.WriteLine(str);

輸出：

One

Two

Three

Five

Six

示例：C＃中的Concat方法

IList<int> collection1 = new List<int>() { 1, 2, 3 };

IList<int> collection2 = new List<int>() { 4, 5, 6 };

var collection3 = collection1.Concat(collection2);

foreach (int i in collection3)

    Console.WriteLine(i);

輸出：

1  
2  
3  
4  
5  
6

C＃或VB.Net中的查詢語法不支援Concat運運算元。

# LINQ 生成運運算元 DefaultIfEmpty

如果調用DefaultIfEmpty()的給定集合為空，則DefaultIfEmpty()方法將返回一個具有預設值的新集合。

DefaultIfEmpty()的另一個重載方法接受一個值參數，該參數應替換為預設值。

看以下示例。

示例：DefaultIfEmpty - C＃

IList<string> emptyList = new List<string>();

var newList1 = emptyList.DefaultIfEmpty();

var newList2 = emptyList.DefaultIfEmpty("None");

Console.WriteLine("Count: {0}" , newList1.Count());

Console.WriteLine("Value: {0}" , newList1.ElementAt(0));

Console.WriteLine("Count: {0}" , newList2.Count());

Console.WriteLine("Value: {0}" , newList2.ElementAt(0));

輸出：

Count: 1

Value:

Count: 1

Value: None

在上面的示例中，emptyList.DefaultIfEmpty() 返回一個新的字串集合，其中一個元素的值為null，因為null是string的預設值。另一種方法emptyList.DefaultIfEmpty("None") 返回一個字串集合，該字串集合的一個元素的值為“ None”而不是null。

下面的示例演示如何在int集合上調用DefaultIfEmpty。

示例：DefaultIfEmpty - C＃

IList<int> emptyList = new List<int>();

var newList1 = emptyList.DefaultIfEmpty();

var newList2 = emptyList.DefaultIfEmpty(100);

Console.WriteLine("Count: {0}" , newList1.Count());

Console.WriteLine("Value: {0}" , newList1.ElementAt(0));

Console.WriteLine("Count: {0}" , newList2.Count());

Console.WriteLine("Value: {0}" , newList2.ElementAt(0));

輸出：

Count: 1

Value: 0

Count: 1

Value: 100

下面的示例演示複雜類型集合的 DefaultIfEmpty() 方法。

示例：DefaultIfEmpty - C＃：

IList<Student> emptyStudentList = new List<Student>();

var newStudentList1 = studentList.DefaultIfEmpty(new Student());

var newStudentList2 = studentList.DefaultIfEmpty(new Student(){

                StudentID = 0,

                StudentName = "" });

Console.WriteLine("Count: {0} ", newStudentList1.Count());

Console.WriteLine("Student ID: {0} ", newStudentList1.ElementAt(0));

Console.WriteLine("Count: {0} ", newStudentList2.Count());

Console.WriteLine("Student ID: {0} ", newStudentList2.ElementAt(0).StudentID);

輸出：

Count: 1

Student ID:

Count: 1

Student ID: 0

# LINQ 生成運運算元 Empty, Range, Repeat

LINQ包括生成運運算元DefaultIfEmpty，Empty，Range＆Repeat。Empty，Range和Repeat方法不是IEnumerable或IQueryable的擴展方法，而只是在靜態類Enumerable中定義的靜態方法。

| **方法** | **描述** |
| --- | --- |
| Empty | 返回一個空集合 |
| Range | 從第一個元素開始，使用指定數量的具有順序值的元素生成IEnumerable <T>類型的集合。 |
| Repeat | 生成具有指定元素數的IEnumerable <T>類型的集合，並且每個元素包含相同的指定值。 |

## Empty

Empty()與其他LINQ方法一樣，該方法不是IEnumerable或IQueryable的擴展方法。它是Enumerable靜態類中包含的靜態方法。因此，您可以像其他靜態方法（如Enumerable.Empty <TResult>()）一樣調用它。Empty()方法返回指定類型的空集合，如下所示。

示例：Enumerable.Empty() 方法

var emptyCollection1 = Enumerable.Empty<string>();

var emptyCollection2 = Enumerable.Empty<Student>();

Console.WriteLine("Count: {0} ", emptyCollection1.Count());

Console.WriteLine("Type: {0} ", emptyCollection1.GetType().Name );

Console.WriteLine("Count: {0} ",emptyCollection2.Count());

Console.WriteLine("Type: {0} ", emptyCollection2.GetType().Name );

輸出：

Type: String[]

Count: 0

Type: Student[]

Count: 0

## Range

Range()方法返回IEnumerable <T>類型的集合，該集合具有指定數量的元素和從第一個元素開始的順序值。

例：Enumerable.Range() 方法使用

var intCollection = Enumerable.Range(10, 10);

Console.WriteLine("總計數: {0} ", intCollection.Count());

for(int i = 0; i < intCollection.Count(); i++)

    Console.WriteLine("值，索引位置為 {0} : {1}", i, intCollection.ElementAt(i));

輸出：

總計數: 10

值，索引位置為 0 : 10

值，索引位置為 1 : 11

值，索引位置為 2 : 12

值，索引位置為 3 : 13

值，索引位置為 4 : 14

值，索引位置為 5 : 15

值，索引位置為 6 : 16

值，索引位置為 7 : 17

值，索引位置為 8 : 18

值，索引位置為 9 : 19

在上面的示例中，Enumerable.Range(10, 10)創建了具有10個整數元素的集合，其順序值從10開始。第一個參數指定元素的起始值，第二個參數指定要創建的元素數。

## Repeat

Repeat()方法使用指定數量的元素生成IEnumerable <T>類型的集合，每個元素包含相同的指定值。

示例：Enumerable.Repeat()方法

var intCollection = Enumerable.Repeat<int>(10, 10);

Console.WriteLine("總數: {0} ", intCollection.Count());

for(int i = 0; i < intCollection.Count(); i++)

    Console.WriteLine("值，索引位置為 {0} : {1}", i, intCollection.ElementAt(i));

輸出：

總數：10

值，索引位置為 0: 10

值，索引位置為 1: 10

值，索引位置為 2: 10

值，索引位置為 3: 10

值，索引位置為 4: 10

值，索引位置為 5: 10

值，索引位置為 6: 10

值，索引位置為 7: 10

值，索引位置為 8: 10

值，索引位置為 9: 10

在上面的示例中，Enumerable.Repeat<int>(10, 10) 創建具有100個重複值為10的整數類型元素的集合，第一個參數指定所有元素的值，第二個參數指定要創建的元素數。

# LINQ Set運運算元 Distinct

下表列出了LINQ中可用的所有Set運運算元。

| **集合運運算元** | **用法** |
| --- | --- |
| Distinct | 返回集合中的非重複值。 |
| Except | 返回兩個序列之間的差，這意味著一個集合中的元素不出現在第二個集合中。 |
| Intersect | 返回兩個序列的交集，即同時出現在兩個集合中的元素。 |
| Union | 返回兩個序列中的唯一元素，這意味著出現在兩個序列中的唯一元素。 |

## Distinct

Distinct擴展方法從給定集合返回一個新的唯一元素集合。

示例：C＃中的Distinct方法

IList<string> strList = new List<string>(){ "One", "Two", "Three", "Two", "Three" };

IList<int> intList = new List<int>(){ 1, 2, 3, 2, 4, 4, 3, 5 };

var distinctList1 = strList.Distinct();

foreach(var str in distinctList1)

    Console.WriteLine(str);

var distinctList2 = intList.Distinct();

foreach(var i in distinctList2)

    Console.WriteLine(i);

輸出：

One

Two

Three

1

2

3

4

5

Distinct擴展方法不比較複雜類型物件的值。為了比較複雜類型的值，需要實現IEqualityComparer<T>介面。在下麵的示例中，StudentComparer類實現IEqualityComparer<Student>來比較Student<objects。

示例：用C＃實現IEqualityComparer

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

    public bool Equals(Student x, Student y)

    {

        if (x.StudentID == y.StudentID

                && x.StudentName.ToLower() == y.StudentName.ToLower())

            return true;

        return false;

    }

    public int GetHashCode(Student obj)

    {

        return obj.StudentID.GetHashCode();

    }

}

現在，您可以在Distinct()方法中傳遞上述StudentComparer類的物件作為參數來比較Student對象，如下所示。 示例：C＃中的Distinct比較物件

IList<Student> studentList = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

var distinctStudents = studentList.Distinct(new StudentComparer());

foreach(Student std in distinctStudents)

    Console.WriteLine(std.StudentName);

輸出：

John

Steve

Bill

Ron

## 查詢語法中的Distinct運運算元

C# 查詢語法不支援 Distinct 運運算元。但是，您可以使用 Distinct 方法查詢變數或將整個查詢包裝到括弧中，然後調用 Distinct ()。

在VB.Net查詢語法中使用Distinct關鍵字：

示例：VB.Net 中 Distinct 的查詢語法

Dim strList = New List(Of string) From {"One", "Three", "Two", "Two", "One" }

Dim distinctStr = From s In strList \_

                  Select s Distinct

# LINQ Set運運算元 Except

Except()方法需要兩個集合。它返回一個新集合，其中包含來自第一個集合的元素，該元素在第二個集合（參數集合）中不存在。

示例：C＃中方法語法 Except

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Five" };

IList<string> strList2 = new List<string>(){"Four", "Five", "Six", "Seven", "Eight"};

var result = strList1.Except(strList2);

foreach(string str in result)

        Console.WriteLine(str);

輸出：

One

Two

Three

Except擴展方法不返回複雜類型集合的正確結果。您需要實現IEqualityComparer介面，以便從Except方法獲得正確的結果。

為 Student 類實現 IEqualityComparer 介面，如下所示:

**示例:  C# 使用 Except 方法的 IEqualityComparer**

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

    public bool Equals(Student x, Student y)

    {

        if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

            return true;

        return false;

    }

    public int GetHashCode(Student obj)

    {

        return obj.StudentID.GetHashCode();

    }

}

現在，您可以在Except擴展方法中通過StudentComparer類，以獲取正確的結果：

**示例：C#物件類型為的Except()方法**

IList<Student> studentList1 = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

IList<Student> studentList2 = new List<Student>() {

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

var resultedCol = studentList1.Except(studentList2,new StudentComparer());

foreach(Student std in resultedCol)

    Console.WriteLine(std.StudentName);

輸出：

John

Steve

C # & VB.Net 查詢語法不支援 Except 運運算元。但是，您可以對查詢變數使用 Distinct 方法，或者將整個查詢包裝到括弧中，然後調用 Except ()。

# LINQ Set運運算元 Intersect

Intersect擴展方法需要兩個集合。它返回一個新集合，其中包含兩個集合中都存在的公共元素。看下麵的實例。

**例如:在方法語法C#中的Intersect**

IList<string> strList1 = new List<string>() { "One", "Two", "Three", "Four", "Five" };

IList<string> strList2 = new List<string>() { "Four", "Five", "Six", "Seven", "Eight"};

var result = strList1.Intersect(strList2);

foreach(string str in result)

        Console.WriteLine(str);

輸出：

Four

Five

Intersect 擴展方法不返回複雜類型集合的正確結果。為了從 Intersect 方法中得到正確的結果，需要實現 IEqualityComparer 介面。

為 Student 類實現 IEqualityComparer 介面，如下所示:

示例：在C＃中將IEqualityComparer與Intersect一起使用

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

    public bool Equals(Student x, Student y)

    {

        if (x.StudentID == y.StudentID &&

                        x.StudentName.ToLower() == y.StudentName.ToLower())

            return true;

        return false;

    }

    public int GetHashCode(Student obj)

    {

        return obj.StudentID.GetHashCode();

    }

}

現在，您可以在Intersect擴展方法中通過StudentComparer類，以獲取正確的結果：

**示例:C#中的Intersect運運算元**

IList<Student> studentList1 = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

IList<Student> studentList2 = new List<Student>() {

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

var resultedCol = studentList1.Intersect(studentList2, new StudentComparer());

foreach(Student std in resultedCol)

    Console.WriteLine(std.StudentName);

輸出：

Bill

Ron

# LINQ Set運運算元 Union

Union擴展方法需要兩個集合，並返回一個新集合，其中包含兩個集合中不同的元素。看下麵的實例。

示例：Union()在C＃中

IList<string> strList1 = new List<string>() { "One", "Two", "three", "Four" };

IList<string> strList2 = new List<string>() { "Two", "THREE", "Four", "Five" };

var result = strList1.Union(strList2);

foreach(string str in result)

        Console.WriteLine(str);

輸出：

One

Two

three

THREE

Four

Five

Union擴展方法不能為複雜類型的集合返回正確的結果。您需要實現IEqualityComparer介面，以便從Union方法中獲取正確的結果。

為 Student 類實現IEqualityComparer介面，如下所示：

**示例: 使用 IEqualityComparer 的 Union 運運算元:**

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

    public bool Equals(Student x, Student y)

    {

        if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

            return true;

        return false;

    }

    public int GetHashCode(Student obj)

    {

        return obj.StudentID.GetHashCode();

    }

}

現在，您可以在Union擴展方法中傳遞上述StudentComparer類以獲得正確的結果：

示例：Uion運運算元 - C＃

IList<Student> studentList1 = new List<Student>() {

        new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

        new Student() { StudentID = 2, StudentName = "Steve",  Age = 15 } ,

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

IList<Student> studentList2 = new List<Student>() {

        new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

        new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

    };

var resultedCol = studentList1.Union(studentList2, new StudentComparer());

foreach(Student std in resultedCol)

    Console.WriteLine(std.StudentName);

輸出：

John

Steve

Bill

Ron

## 查詢語法

C # & VB.Net 查詢語法不支援聯合運運算元。但是，您可以對查詢變數使用 Union 方法，或者將整個查詢包裝到方括號中，然後調用 Union ()。

# LINQ 分區運運算元 Skip & SkipWhile

分區運運算元將序列（集合）分為兩部分，並返回其中一部分。

| **方法** | **描述** |
| --- | --- |
| Skip | 從序列中的第一個元素開始，將元素跳到指定的位置。 |
| SkipWhile | 根據條件跳過元素，直到元素不滿足條件為止。如果第一個元素本身不滿足條件，那麼它將跳過0個元素並返回序列中的所有元素。 |
| Take | 從序列中的第一個元素開始，將元素帶到指定的位置。 |
| TakeWhile | 從第一個元素返回元素，直到元素不滿足條件為止。如果第一個元素本身不滿足條件，則返回一個空集合。 |

## Skip

Skip()方法從第一個元素開始跳過指定數量的元素，並返回其餘元素。

示例：Skip()方法跳過2個元素 -C＃

IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };

var newList = strList.Skip(2);

foreach(var str in newList)

    Console.WriteLine(str);

輸出：

Three

Four

Five

## 查詢語法中的 Skip 運運算元

C # 查詢語法不支援 Skip & SkipWhile 運運算元。但是，您可以對查詢變數使用 Skip/SkipWhile 方法，或者將整個查詢包裝到括弧中，然後調用 Skip/SkipWhile。

下面的示例演示了查詢語法中的skip運運算元-VB.NET版

**示例：VB.Net中的Skip運運算元**

Dim skipResult = From s In studentList

                 Skip 3

                 Select s

## SkipWhile

顧名思義，LINQ中的SkipWhile()擴展方法將跳過集合中的元素，直到指定的條件為true。一旦任何元素的指定條件變為false，它將返回一個包含所有剩餘元素的新集合。

SkipWhile()方法有兩種重載方法。一種方法接受Func<TSource, bool>類型的謂詞，另一種重載方法接受Func<TSource, int, bool>通過元素索引的謂詞類型。

在下面的示例中，SkipWhile()方法跳過所有元素，直到找到長度等於或大於4個字元的字串。

**示例：在C＃中SkipWhile()方法跳過長度小於4的元素,並返回其後的所有元素**

IList<string> strList = new List<string>() {

                                            "One",

                                            "Two",

                                            "Three",

                                            "Four",

                                            "Five",

                                            "Six"  };

var resultList = strList.SkipWhile(s => s.Length < 4);

foreach(string str in resultList)

        Console.WriteLine(str);

輸出：

Three

Four

Five

Six

在上面的示例中，由於前兩個元素的長度小於3，所以 SkipWhile() 跳過前兩個元素，並找到長度等於或大於4的第三個元素。一旦找到長度等於或大於4個字元的任何元素，它將不跳過其他任何元素元素，即使小於4個字元。

現在，看以下示例，該示例中，其中 SkipWhile ()不會跳過任何元素，因為第一個元素的指定條件為 false。

**示例：C＃使用SkipWhile方法，不跳過元素**

IList<string> strList = new List<string>() {

                                            "Three",

                                            "One",

                                            "Two",

                                            "Four",

                                            "Five",

                                            "Six"  };

var resultList = strList.SkipWhile(s => s.Length < 4);

foreach(string str in resultList)

        Console.WriteLine(str);

輸出：

Three

One

Two

Four

Five

Six

SkipWhile的第二個重載傳遞每個元素的索引。看下麵的實例

**示例:在C#中使用索引的SkipWhile()方法**

IList<string> strList = new List<string>() {

                                            "One",

                                            "Two",

                                            "Three",

                                            "Four",

                                            "Five",

                                            "Six"  };

var result = strList.SkipWhile((s, i) => s.Length > i);

foreach(string str in result)

    Console.WriteLine(str);

輸出：

Five

Six

在上面的示例中，lambda運算式包括元素和元素的索引作為參數。它會跳過所有元素，直到字串元素的長度大於其索引。

## 查詢語法中的SkipWhile運運算元

C# 查詢語法不支援 Skip & SkipWhile 運運算元。但是，您可以對查詢變數使用 Skip/SkipWhile 方法，或者將整個查詢包裝到括弧中，然後調用 Skip/SkipWhile ()。

示例：VB.Net中的SkipWhile方法

Dim strList = New List(Of string) From {

                                        "One",

                                        "Two",

                                        "Three",

                                        "Four",

                                        "Five",

                                        "Six"  }

Dim skipWhileResult = From s In studentList

                      Skip While s.Length < 4

                      Select s

輸出：

Three

Four

Five

Six

# LINQ 分區運運算元 Take & TakeWhile

分區運運算元將序列（集合）分為兩部分，並返回其中一部分。

## Take

Take()擴展方法返回從第一個元素開始的指定數量的元素。

示例：Take()方法在C＃中

IList<string> strList = new List<string>(){ "One", "Two", "Three", "Four", "Five" };

var newList = strList.Take(2);

foreach(var str in newList)

    Console.WriteLine(str);

輸出：

One

Two

C# 查詢語法不支援 Take & takedwhile 運運算元。但是，您可以對查詢變數使用 Take/takedwhile 方法，或者將整個查詢包裝到括弧中，然後調用 Take/takedwhile ()。

**示例：VB.Net查詢語法中 使用 Take 運運算元**

Dim takeResult = From s In studentList

                 Take 3

                 Select s

## TakeWhile

TakeWhile()擴展方法返回給定集合中的元素，直到指定的條件為true。如果第一個元素本身不滿足條件，則返回一個空集合。

TakeWhile方法有兩個重載方法。一種方法接受Func<TSource, bool>類型的謂詞，另一種重載方法接受Func<TSource, int, bool>通過元素索引的謂詞類型。

在下面的示例中，TakeWhile()方法返回一個包含所有元素的新集合，直到找到長度小於4個字元的字串為止。

示例：C＃中的TakeWhile方法返回字串長度大於4的元素

IList<string> strList = new List<string>() {

                                            "Three",

                                            "Four",

                                            "Five",

                                            "Hundred"  };

var result = strList.TakeWhile(s => s.Length > 4);

foreach(string str in result)

        Console.WriteLine(str);

輸出：

Three

在上面的示例中，TakeWhile()返回僅包含第一元素，因為第二字串元素不滿足該條件。

TakeWhile 還在謂詞函數中傳遞當前元素的索引。下面的 TakeWhile 方法示例接受元素，直到字串元素的長度大於它的索引

**示例：C#中的TakeWhile傳遞索引,返回字串長度大於索引的元素**

IList<string> strList = new List<string>() {

                                            "One",

                                            "Two",

                                            "Three",

                                            "Four",

                                            "Five",

                                            "Six"  };

var resultList = strList.TakeWhile((s, i) => s.Length > i);

foreach(string str in resultList)

        Console.WriteLine(str);

輸出：

One

Two

Three

Four

# LINQ 轉換運運算元

LINQ中的Conversion運運算元可用於轉換序列（集合）中元素的類型。轉換運運算元分為三種：**As**運運算元（AsEnumerable和AsQueryable），**To**運運算元（ToArray，ToDictionary，ToList和ToLookup）和**轉換**運運算元（Cast和OfType）。

下表列出了所有轉換運運算元。

| **方法** | **描述** |
| --- | --- |
| AsEnumerable | 將輸入序列作為 IEnumerable < T> 返回 |
| AsQueryable | 將IEnumerableto轉換為IQueryable，以類比遠端查詢提供程式 |
| Cast | 將非泛型集合轉換為泛型集合（IEnumerable到IEnumerable） |
| OfType | 基於指定類型篩選集合 |
| ToArray | 將集合轉換為陣列 |
| ToDictionary | 根據鍵選擇器函數將元素放入 Dictionary 中 |
| ToList | 將集合轉換為 List |
| ToLookup | 將元素分組到 Lookup<TKey,TElement> |

## AsEnumerable和AsQueryable方法

AsEnumerable和AsQueryable方法分別將源物件轉換或轉換為IEnumerable <T>或IQueryable <T>。

請看以下示例：

示例：C＃中的AsEnumerable和AsQueryable運運算元：

class Program

{

    static void ReportTypeProperties<T>(T obj)

    {

        Console.WriteLine("Compile-time type: {0}", typeof(T).Name);

        Console.WriteLine("Actual type: {0}", obj.GetType().Name);

    }

    static void Main(string[] args)

    {

        Student[] studentArray = {

                new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

                new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

                new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

                new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 } ,

            };

        ReportTypeProperties( studentArray);

        ReportTypeProperties(studentArray.AsEnumerable());

        ReportTypeProperties(studentArray.AsQueryable());

    }

}

輸出：

Compile-time type: Student[]

Actual type: Student[]

Compile-time type: IEnumerable`1

Actual type: Student[]

Compile-time type: IQueryable`1

Actual type: EnumerableQuery`1

如上例所示，AsEnumerable和AsQueryable方法分別將編譯時間類型轉換為IEnumerable和IQueryable

## Cast

Cast的作用與AsEnumerable<T>相同。它將源物件強制轉換為IEnumerable<T>。

示例：C＃中的 Cast轉換運運算元

class Program

{

    static void ReportTypeProperties<T>(T obj)

    {

        Console.WriteLine("Compile-time type: {0}", typeof(T).Name);

        Console.WriteLine("Actual type: {0}", obj.GetType().Name);

    }

    static void Main(string[] args)

    {

        Student[] studentArray = {

                new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

                new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

                new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,

                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

                new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 } ,

            };

        ReportTypeProperties( studentArray);

        ReportTypeProperties(studentArray.Cast<Student>());

    }

}

輸出：

Compile-time type: Student[]

Actual type: Student[]

Compile-time type: IEnumerable`1

Actual type: Student[]

Compile-time type: IEnumerable`1

Actual type: Student[]

Compile-time type: IEnumerable`1

Actual type: Student[]

studentArray.Cast<Student>() 與 (IEnumerable<Student>)studentArray 相同，但是 Cast<Student>() 可讀性更好。

## To運運算元：ToArray()，ToList()，ToDictionary()

顧名思義，ToArray()，ToList()，ToDictionary()方法的源物件轉換分別為一個陣列，清單或字典。

To 運運算元強制執行查詢。它強制遠端查詢提供者執行查詢並從底層資料來源(如SQL Server資料庫)獲取結果。

示例：C＃中的ToArray和ToList

IList<string> strList = new List<string>() {

                                            "One",

                                            "Two",

                                            "Three",

                                            "Four",

                                            "Three"

                                            };

string[] strArray = strList.ToArray<string>();// 將清單轉換為陣列

IList<string> list = strArray.ToList<string>(); // converts array into list

ToDictionary - 將泛型列表轉換為泛型詞典：

示例：C＃中的 ToDictionary：

IList<Student> studentList = new List<Student>() {

                    new Student() { StudentID = 1, StudentName = "John", age = 18 } ,

                    new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,

                    new Student() { StudentID = 3, StudentName = "Bill",  age = 18 } ,

                    new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,

                    new Student() { StudentID = 5, StudentName = "Ron" , age = 21 }

                };

//以下將清單轉換成字典，其中StudentId是鍵

IDictionary<int, Student> studentDict =

                                studentList.ToDictionary<Student, int>(s => s.StudentID);

foreach(var key in studentDict.Keys)

Console.WriteLine("Key: {0}, Value: {1}",

                                key, (studentDict[key] as Student).StudentName);

輸出：

Key: 1, Value: John

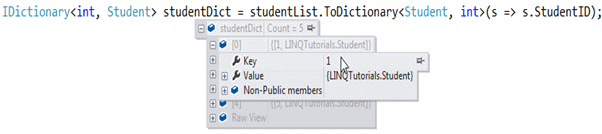
Key: 2, Value: Steve

Key: 3, Value: Bill

Key: 4, Value: Ram

Key: 5, Value: Ron

下圖顯示了上面示例中的studentDict如何包含一個key-value對，其中key是StudentID，value是Student對象。

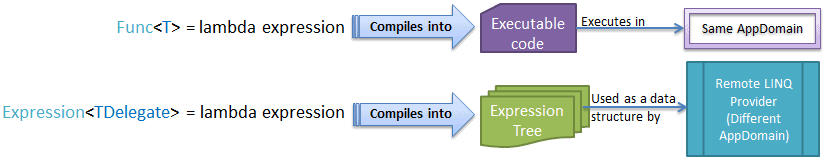
LINQ-ToDictionary運運算元

# LINQ 運算式（Expression）

可以將lambda運算式分配給Func或Action類型委託，以處理記憶體中的集合。.NET編譯器在編譯時將分配給Func或Action類型委託的lambda運算式轉換為可執行代碼。

LINQ引入了一種名為Expression的新類型，該類型代表強類型的lambda運算式。這意味著lambda運算式也可以分配給Expression <TDelegate>類型。.NET編譯器將分配給Expression <TDelegate>的lambda運算式轉換為Expression樹，而不是可執行代碼。遠端LINQ查詢提供程式使用此運算式樹作為資料結構，以其構建運行時查詢（例如LINQ-to-SQL，EntityFramework或實現IQueryable <T>介面的任何其他LINQ查詢提供程式）。

下圖說明瞭將lambda運算式分配給Func或Action委託與LINQ中的Expression時的區別。

Expression 和 Func

在下一節中，我們將學習運算式樹，但首先，讓我們看看如何定義和調用運算式。

## 定義 Expression

引用System.Linq.Expressions命名空間，並使用Expression <TDelegate>類定義一個Expression。Expression <TDelegate>需要委託類型Func或Action。

例如，你可以將 lambda 運算式賦給 Func 類型委託 的 isTeenAger 變數，如下所示:

示例：在C＃中為運算式定義Func委託

public class Student

{

    public int StudentID { get; set; }

    public string StudentName { get; set; }

    public int Age { get; set; }

}

Func<Student, bool> isTeenAger = s => s.Age > 12 && s.Age < 20;

示例：在VB.Net中為運算式定義Func委託

Dim isTeenAger As Func(Of Student, Boolean) = Function(s) s.Age > 12 And s.Age < 20

現在，您可以使用Expresson包裝Func委託，將上述Func類型委託轉換為Expression，如下所示：

 示例：在C#中定義運算式 Expresson

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

示例：在VB.Net中定義運算式

Dim isTeenAgerExpr As Expression(Func(Of Student, Boolean)) =

                                        Function(s) s.Age > 12 And s.Age < 20

以相同的方式，如果您不從委託返回值，則還可以用Expression包裝Action <t>類型委託。

示例：在C＃中定義運算式

Expression<Action<Student>> printStudentName = s => Console.WriteLine(s.StudentName);

示例：在VB.Net中定義運算式

Dim printStudentName As Expression(Action(Of Student) =

                                        Function(s) Console.WriteLine(s.StudentName);

因此，您可以定義Expression <TDelegate>類型。現在，讓我們看看如何調用由Expression <TDelegate>包裝的委託。

## 調用運算式(Expression)

您可以用與委託相同的方式調用由Expression包裹的委託，但是首先需要使用Compile()方法進行編譯。Compile()返回**Func**或**Action**類型的委託，以便您可以像委託一樣調用它。

    示例：在C＃中調用運算式

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

//使用Compile方法編譯Expression以將其作為委託調用

Func<Student, bool>  isTeenAger = isTeenAgerExpr.Compile();

//Invoke

bool result = isTeenAger(new Student(){ StudentID = 1, StudentName = "Steve", Age = 20});

示例：在VB.Net中調用運算式

Dim isTeenAgerExpr As Expression(Of Func(Of Student, Boolean)) =

                                                    Function(s) s.Age > 12 And s.Age < 20

'使用compile方法將其作為委託調用的compile運算式

Dim isTeenAger As Func(Of Student, Boolean) = isTeenAgerExpr.Compile()

Dim result = isTeenAger(New Student() With { .StudentID = 1, .StudentName = "Steve", .Age = 20})

在下一節中將詳細瞭解運算式樹。

# LINQ 運算式樹

您已在上一節中瞭解了運算式。現在，讓我們在這裡瞭解運算式樹。

顧名思義，運算式樹不過是按樹狀資料結構排列的運算式。運算式樹中的每個節點都是一個運算式。例如，運算式樹可用於表示數學公式x <y，其中x，<和y將表示為運算式，並排列在樹狀結構中。

運算式樹是lambda運算式的記憶體表示形式。它保存查詢的實際元素，而不是查詢的結果。

運算式樹使lambda運算式的結構透明和顯式。您可以與運算式樹中的資料進行交互，就像與其他任何資料結構一樣。

例如，看以下isTeenAgerExpr運算式：

**示例：用C#運算式**

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.age > 12 && s.age < 20;

編譯器會將上面的運算式轉換為以下運算式樹：

**示例：C＃中的運算式樹**

Expression.Lambda<Func<Student, bool>>(

                Expression.AndAlso(

                    Expression.GreaterThan(Expression.Property(pe, "Age"), Expression.Constant(12, typeof(int))),

                    Expression.LessThan(Expression.Property(pe, "Age"), Expression.Constant(20, typeof(int)))),

                        new[] { pe });

您也可以手動構建運算式樹。讓我們看看如何為以下簡單的lambda運算式構建運算式樹：

**示例：C＃中的Func委託：**

Func<Student, bool> isAdult = s => s.age >= 18;

此Func類型委託將被視為以下方法：

 C＃：

public bool function(Student s)

{

  return s.Age > 18;

}

要創建運算式樹，首先，創建一個參數運算式，其中Student是參數的類型，'s'是參數的名稱，如下所示：

**步驟1：在C＃中創建參數運算式**

ParameterExpression pe = Expression.Parameter(typeof(Student), "s");

現在，使用Expression.Property()創建s.Age運算式，其中s是參數，Age是Student的屬性名稱。（**Expression**是一個抽象類別，其中包含用於手動創建運算式樹的靜態説明器方法。）

**步驟2：在C＃中創建屬性運算式**

MemberExpression me = Expression.Property(pe, "Age");

現在，為18創建一個常量運算式：

**步驟3：在C＃中創建常量運算式**

ConstantExpression constant = Expression.Constant(18, typeof(int));

到目前為止，我們已經為s.Age（成員運算式）和18（常量運算式）構建了運算式樹。現在，我們需要檢查成員運算式是否大於常量運算式。為此，請使用Expression.GreaterThanOrEqual() 方法，並將成員運算式和常量運算式作為參數傳遞：：

**步驟4：在C＃中創建二進位運算式**

BinaryExpression body = Expression.GreaterThanOrEqual(me, constant);

因此，我們為lambda運算式主體 s.Age> = 18 構建了一個運算式樹。我們現在需要將參數運算式和主體運算式連接起來。使用Expression.Lambda(body,parameters array)連接lambda運算式s => s.age> = 18的body(主體)和parameter(參數)部分：

**步驟5：在C＃中創建Lambda運算式**

var isAdultExprTree = Expression.Lambda<Func<Student, bool>>(body, new[] { pe });

這樣，您可以為帶有lambda運算式的簡單Func委託構建運算式樹。

**示例：C＃中的運算式樹**

ParameterExpression pe = Expression.Parameter(typeof(Student), "s");

MemberExpression me = Expression.Property(pe, "Age");

ConstantExpression constant = Expression.Constant(18, typeof(int));

BinaryExpression body = Expression.GreaterThanOrEqual(me, constant);

var ExpressionTree = Expression.Lambda<Func<Student, bool>>(body, new[] { pe });

Console.WriteLine("運算式樹: {0}", ExpressionTree);

Console.WriteLine("運算式樹體: {0}", ExpressionTree.Body);

Console.WriteLine("運算式樹中的參數個數: {0}",

                                ExpressionTree.Parameters.Count);

Console.WriteLine("運算式樹中的參數: {0}", ExpressionTree.Parameters[0]);

示例：VB.Net中的表達樹

Dim pe As ParameterExpression = Expression.Parameter(GetType(Student), "s")

Dim mexp As MemberExpression = Expression.Property(pe, "Age")

Dim constant As ConstantExpression = Expression.Constant(18, GetType(Integer))

Dim body As BinaryExpression = Expression.GreaterThanOrEqual(mexp, constant)

Dim ExpressionTree As Expression(Of Func(Of Student, Boolean)) =

    Expression.Lambda(Of Func(Of Student, Boolean))(body, New ParameterExpression() {pe})

Console.WriteLine("運算式樹: {0}", ExpressionTree)

Console.WriteLine("運算式樹體: {0}", ExpressionTree.Body)

Console.WriteLine("運算式樹中的參數個數: {0}",

                                ExpressionTree.Parameters.Count)

Console.WriteLine("運算式樹中的參數: {0}", ExpressionTree.Parameters(0))

輸出：

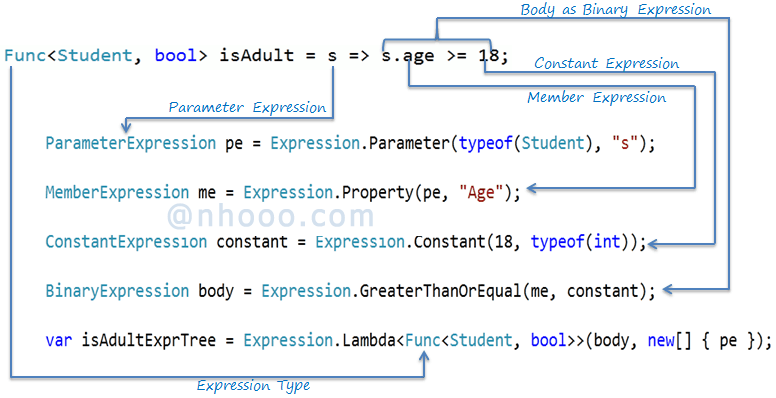
運算式樹: s => (s.Age >= 18)

運算式樹體: (s.Age >= 18)

運算式樹中的參數個數: 1

運算式樹中的參數: s

下圖說明瞭創建運算式樹的整個過程：

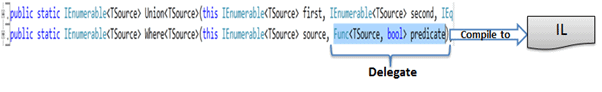
構造運算式樹

## 為什麼選擇表達樹？

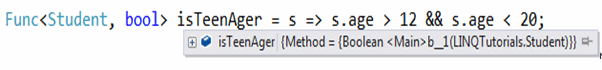
在上一節中，我們已經看到分配給lambda運算式**Func<T>**編譯為可執行代碼，分配給lambda運算式**Expression<TDelegate>**類型編譯為Expression樹。

可執行代碼在同一個應用程式定義域中執行，以處理記憶體中的集合。可枚舉的靜態類包含用於實現**IEnumerable <T>**介面的記憶體中集合的擴展方法，例如List <T>，Dictionary <T>等。Enumerable類中的擴展方法接受**Func**類型委託的謂詞參數。例如，**Where**擴展方法接受**Func <TSource，bool>謂詞**。然後，將其編譯為IL（中間語言）以處理同一AppDomain中的記憶體中集合。

下圖顯示了Enumerable類中的where擴展方法包括Func委託作為參數的情況：

Where 的Func 委託

**Func**委託是原始的可執行代碼，因此，如果調試代碼，則會發現**Func**委託將表示為不透明代碼。您無法看到其參數，返回類型和主體：

調試模式下的Func委託

**Func**委託用於記憶體中的集合，因為它將在同一個AppDomain中進行處理，但是諸如LINQ-to-SQL，EntityFramework或其他提供LINQ功能的協力廠商產品的遠端LINQ查詢提供者呢？他們將如何解析已編譯為原始可執行代碼的lambda運算式，以瞭解參數，lambda運算式的返回類型以及構建運行時查詢以進一步處理？答案是**表達樹**。

Expression <TDelegate>被編譯成稱為運算式樹的資料結構。

如果調試代碼，則運算式代表將如下所示：

調試模式下的運算式樹

現在您可以看到普通委託和運算式之間的區別。運算式樹是透明的。您可以從運算式中檢索參數，返回類型和主體運算式資訊，如下所示：

示例：C＃中的運算式樹

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

Console.WriteLine("Expression: {0}", isTeenAgerExpr );

Console.WriteLine("運算式類型: {0}", isTeenAgerExpr.NodeType);

var parameters = isTeenAgerExpr.Parameters;

foreach (var param in parameters)

{

    Console.WriteLine("參數名稱: {0}", param.Name);

    Console.WriteLine("參數類型: {0}", param.Type.Name );

}

var bodyExpr = isTeenAgerExpr.Body as BinaryExpression;

Console.WriteLine("運算式主體左側: {0}", bodyExpr.Left);

Console.WriteLine("二進位運算式類型: {0}", bodyExpr.NodeType);

Console.WriteLine("運算式主體右側: {0}", bodyExpr.Right);

Console.WriteLine("返回類型: {0}", isTeenAgerExpr.ReturnType);

輸出：

Expression: s => ((s.Age > 12) AndAlso (s.Age < 20))

運算式類型: Lambda

參數名稱: s

參數類型: Student

運算式主體左側: (s.Age > 12)

二進位運算式類型: AndAlso

運算式主體右側: (s.Age < 20)

返回類型: System.Boolean

不在同一應用程式定義域中執行針對LINQ-to-SQL或Entity Framework的LINQ查詢。例如，對於Entity Framework的以下LINQ查詢永遠不會在程式內部實際執行：

示例：C＃中的LINQ查詢

var query = from s in dbContext.Students

            where s.Age >= 18

            select s;

首先將其轉換為SQL語句，然後在資料庫伺服器上執行。

在查詢運算式中找到的代碼必須轉換為SQL查詢，該查詢可以作為字串發送到另一個進程。對於LINQ-to-SQL或Entity Framework，該過程恰好是SQL Server資料庫。將資料結構（如運算式樹）轉換為SQL顯然比將原始IL或可執行代碼轉換為SQL容易得多，因為正如您看到的，從運算式中檢索資訊很容易。

創建運算式樹的目的是將諸如查詢運算式之類的代碼轉換為可以傳遞給其他進程並在此處執行的字串。

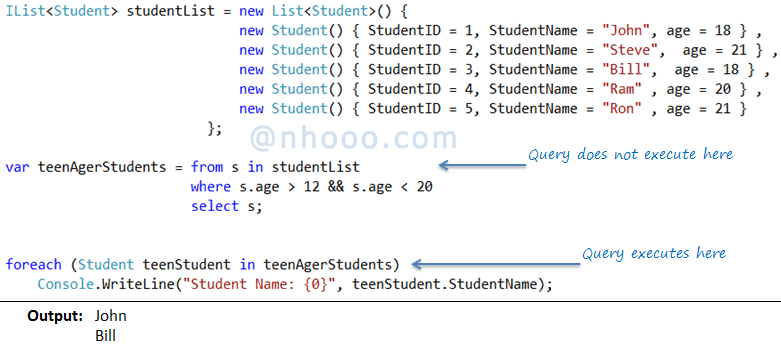
可查詢的靜態類包括接受Expression類型的謂詞參數的擴展方法。將該謂詞運算式轉換為運算式樹，然後將其作為資料結構傳遞到遠端LINQ提供程式，以便提供程式可以從運算式樹構建適當的查詢並執行查詢。

# LINQ 延遲執行查詢

延遲執行意味著運算式的求值被延遲到實際需要的時候。它通過避免不必要的執行，極大地提高了性能。

延遲執行適用於任何記憶體集合以及遠端LINQ提供程式，例如LINQ-to-SQL，LINQ-to-Entities，LINQ-to-XML等。

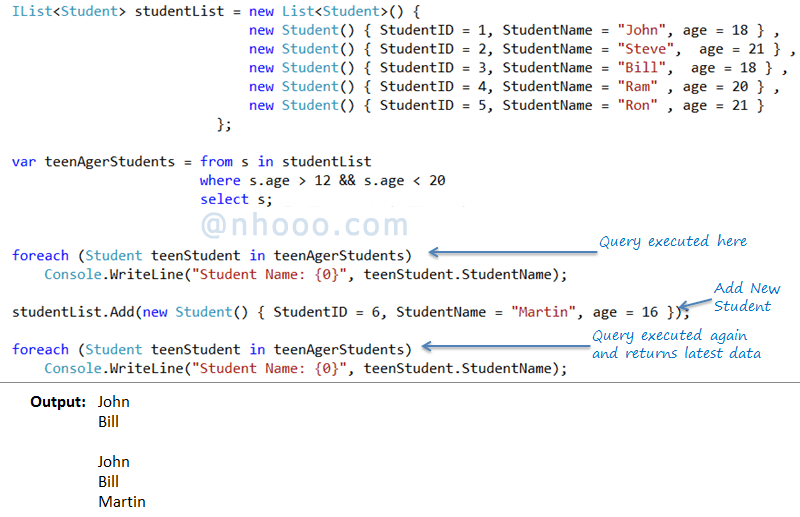
讓我們使用以下示例瞭解延遲執行：

延遲執行

在上面的示例中，您可以看到在使用foreach迴圈進行反覆運算時查詢已實現並執行。這稱為延遲執行。當您實際訪問集合中的每個物件並對其進行處理時，LINQ會處理studentList集合。

## 延遲執行將返回最新資料

要檢查每次延遲執行是否每次都返回最新資料，請在foreach迴圈後再添加一名青少年Ager學生，並檢查青少年學生列表：

延遲執行

正如您看到的，第二個foreach迴圈再次執行查詢並返回最新資料。延遲執行在每次執行時重新計算;這被稱為**惰性求值**。這是延遲執行的主要優點之一:它總是為您提供最新的資料。

## 實現延遲執行

您可以使用c#的 **yield**關鍵字為IEnumerable的自訂擴展方法實現延遲執行。

例如，您可以為IEnumerable實現自訂擴展方法GetTeenAgerStudents，該方法將返回所有青少年學生的列表。

示例：在C＃中實現延遲執行

public static class EnumerableExtensionMethods

{

    public static IEnumerable<Student> GetTeenAgerStudents(this IEnumerable<Student> source)

    {

        foreach (Student std in source)

        {

            Console.WriteLine("Accessing student {0}", std.StudentName);

            if (std.age > 12 && std.age < 20)

                yield return std;

        }

    }

}

請注意，每當GetTeenAgerStudents()被調用時，我們都會在控制台上列印學生姓名。

現在，您可以使用以下擴展方法：

    C＃：

IList<Student> studentList = new List<Student>() {

            new Student() { StudentID = 1, StudentName = "John", age = 13 } ,

            new Student() { StudentID = 2, StudentName = "Steve",  age = 15 } ,

            new Student() { StudentID = 3, StudentName = "Bill",  age = 18 } ,

            new Student() { StudentID = 4, StudentName = "Ram" , age = 12 } ,

            new Student() { StudentID = 5, StudentName = "Ron" , age = 21 }

        };

var teenAgerStudents = from s in studentList.GetTeenAgerStudents()

                        select s;

foreach (Student teenStudent in teenAgerStudents)

    Console.WriteLine("Student Name: {0}", teenStudent.StudentName);

輸出：

Accessing student John

Student Name: John

Accessing student Steve

Student Name: Steve

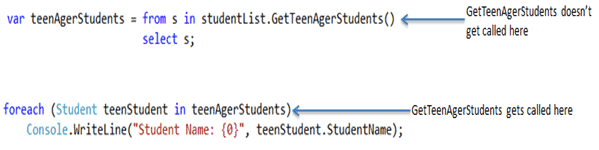
Accessing student Bill

Student Name: Bill

Accessing student Ram

Accessing student Ron

從輸出中可以看到，GetTeenAgerStudents()當您使用foreach迴圈反覆運算studentList時，將被調用。

延遲執行

因此，通過這種方式，您可以使用**yield**關鍵字創建自訂方法，以獲得延遲執行的優勢。

# LINQ 立即執行查詢

立即執行與延遲執行相反。它強制LINQ查詢執行並立即獲取結果。“ To”轉換運運算元執行給定的查詢並立即給出結果。

## 方法語法

在下面的示例中，ToList()擴展方法立即執行查詢並返回結果。

**C＃：立即執行**

IList<Student> teenAgerStudents =

                studentList.Where(s => s.age > 12 && s.age < 20).ToList();

**VB.Net：立即執行**

Dim teenAgerStudents As IList(Of Student) =

                    studentList.Where(Function(s) s.Age > 12 And s.Age < 20).ToList()

## 查詢語法

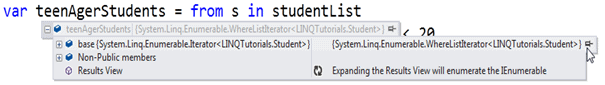
C＃：

var teenAgerStudents = from s in studentList

                where s.age > 12 && s.age < 20

                select s;

上面的查詢不會立即執行。您不會找到任何結果，如下所示：

立即執行

查詢語法不支援“To”運運算元，但可以使用ToList()、ToArray()或ToDictionary()立即執行，如下所示：

C＃：

IList<Student> teenAgerStudents = (from s in studentList

                where s.age > 12 && s.age < 20

                select s).ToList();

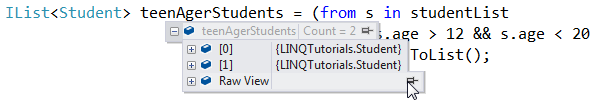
VB.Net：

Dim teenAgerStudents As IList(Of Student) = (From s In studentList \_

                Where s.Age > 12 And s.Age < 20 \_

                Select s).ToList()

您可以在teenAgerStudents集合中查看結果，如下所示：

立即執行

# LINQ let 關鍵字

'let'關鍵字在查詢語法中很有用。它投影了一個新的範圍變數，允許複用使用運算式並使查詢更具可讀性。

例如，您可以比較字串值並選擇小寫字串值，如下所示：

示例：C＃LINQ查詢 let關鍵字

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

var lowercaseStudentNames = from s in studentList

                            where s.StudentName.ToLower().StartsWith("r")

                            select s.StudentName.ToLower();

正如您看到的，ToLower()方法在上面的查詢中被多次使用。下面的示例使用“ let”引入新的變數“ lowercaseStudentName”，然後將在每個地方使用該變數。因此，**let**關鍵字使查詢更具可讀性。

**示例：C＃中的let關鍵字**

var lowercaseStudentNames = from s in studentList

                            let lowercaseStudentName = s.StudentName.ToLower()

                                where lowercaseStudentName.StartsWith("r")

                                select lowercaseStudentName;

foreach (var name in lowercaseStudentNames)

Console.WriteLine(name);

**示例：VB.Net中的let關鍵字**

Dim lowercaseStudentNames = From s In studentList

                            Let lowercaseStudentName = s.StudentName.ToLower()

                                Where lowercaseStudentName.StartsWith("r")

                                Select lowercaseStudentName;

輸出：

ram

ron

# LINQ into 關鍵字

在LINQ查詢中使用into關鍵字可組成一個組或在select子句後繼續查詢。

示例：LINQ中的 into 關鍵字

var teenAgerStudents = from s in studentList

    where s.age > 12 && s.age < 20

    select s

        into teenStudents

        where teenStudents.StudentName.StartsWith("B")

        select teenStudents;

在上面的查詢中，“into”關鍵字引入了一個新的範圍變數teenStudents，因此第一個範圍變數s超出範圍。您可以使用新的範圍變數在into關鍵字之後編寫進一步的查詢。

VB.Net中的'into'關鍵字用於分組。

示例：VB.Net LINQ中的into關鍵字

Dim groupQuery = From s In studentList

                 Group By s.Age Into Group

# LINQ 查詢示例

在本節中，您將學習一些複雜的LINQ查詢。我們將使用以下學生和標準集合進行查詢。

樣本集合：

IList<Student> studentList = new List<Student>() {

    new Student() { StudentID = 1, StudentName = "John", Age = 18, StandardID = 1 } ,

    new Student() { StudentID = 2, StudentName = "Steve",  Age = 21, StandardID = 1 } ,

    new Student() { StudentID = 3, StudentName = "Bill",  Age = 18, StandardID = 2 } ,

    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID = 2 } ,

    new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

IList<Standard> standardList = new List<Standard>() {

    new Standard(){ StandardID = 1, StandardName="Standard 1"},

    new Standard(){ StandardID = 2, StandardName="Standard 2"},

    new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

## 多個Select和where運運算元

    示例：多個Select和where運運算元

var studentNames = studentList.Where(s => s.Age > 18)

                              .Select(s => s)

                              .Where(st => st.StandardID > 0)

                              .Select(s => s.StudentName);

輸出：

Steve

Ram

以下查詢返回僅具有StudentName屬性的匿名物件的Enumerable：

示例：LINQ查詢返回匿名物件的集合

var teenStudentsName = from s in studentList

                       where s.age > 12 && s.age < 20

                       select new { StudentName = s.StudentName };

teenStudentsName.ToList().ForEach(s => Console.WriteLine(s.StudentName));

輸出：

John

Bill

## Group By

以下查詢返回按StandardID列出的學生組：

示例：LINQ Group通過查詢-C＃

var studentsGroupByStandard = from s in studentList

                              group s by s.StandardID into sg

                              orderby sg.Key

                                    select new { sg.Key, sg };

foreach (var group in studentsGroupByStandard)

{

    Console.WriteLine("StandardID {0}:", group.Key);

    group.sg.ToList().ForEach(st => Console.WriteLine(st.StudentName ));

}

輸出：

StandardID 0:

Ron

StandardID 1:

John

Steve

StandardID 2:

Bill

Ram

輸出包括沒有任何 StandardID 的 Ron，因此 Ron 屬於 StandardID 0。

要刪除沒有StandardID的學生，請在組運運算元之前使用where運運算元：

示例：LINQ Group by查詢 - C＃

var studentsGroupByStandard = from s in studentList

                              where s.StandardID > 0

                              group s by s.StandardID into sg

                              orderby sg.Key

                                    select new { sg.Key, sg };

輸出：

StandardID 1:

John

Steve

StandardID 2:

Bill

Ram

## Left outer join

使用左外部聯結(Left outer join)顯示每個標準下的學生。即使沒有分配該標準的學生，也要顯示標準名稱。

示例：C＃LINQ 中Left outer join聯接-

var studentsGroup = from stad in standardList

                    join s in studentList

                    on stad.StandardID equals s.StandardID

                        into sg

                        select new {

                                        StandardName = stad.StandardName,

                                        Students = sg

                                    };

foreach (var group in studentsGroup)

{

    Console.WriteLine(group.StandardName);

    group.Students.ToList().ForEach(st => Console.WriteLine(st.StudentName));

}

輸出：

Standard 1:

John

Steve

Standard 2:

Bill

Ram

Standard 3:

在下麵的group by查詢示例中，我們對組進行排序並只選擇StudentName:

示例：LINQ 左外部聯結 - C＃

var studentsWithStandard = from stad in standardList

                           join s in studentList

                           on stad.StandardID equals s.StandardID

                           into sg

                               from std\_grp in sg

                               orderby stad.StandardName, std\_grp.StudentName

                               select new {

                                                StudentName = std\_grp.StudentName,

                                                StandardName = stad.StandardName

                                };

foreach (var group in studentsWithStandard)

{

    Console.WriteLine("{0} is in {1}", group.StudentName, group.StandardName);

}

輸出：

John is in Standard 1

Steve is in Standard 1

Bill is in Standard 2

Ram is in Standard 2

## 排序

以下查詢按StandardID和Age的昇冪返回學生列表。

示例：排序

var sortedStudents = from s in studentList

                        orderby s.StandardID, s.age

                        select new {

                                StudentName = s.StudentName,

                                Age = s.age,

                                StandardID = s.StandardID };

sortedStudents.ToList().ForEach(s => Console.WriteLine("Student Name: {0}, Age: {1}, StandardID: {2}", s.StudentName, s.Age , s.StandardID));

輸出：

Student Name: Ron, Age: 21, StandardID: 0

Student Name: John, Age: 18, StandardID: 1

Student Name: Steve, Age: 21, StandardID: 1

Student Name: Bill, Age: 18, StandardID: 2

Student Name: Ram, Age: 20, StandardID: 2

## 內部聯接(Inner Join)

示例：LINQ內部聯接-C＃

var studentWithStandard = from s in studentList

                          join stad in standardList

                          on s.StandardID equals stad.StandardID

                          select new {

                                  StudentName = s.StudentName,

                                  StandardName = stad.StandardName

                              };

studentWithStandard.ToList().ForEach(s => Console.WriteLine("{0} is in {1}", s.StudentName, s.StandardName  ));

輸出：

John is in Standard 1

Steve is in Standard 1

Bill is in Standard 2

Ram is in Standard 2

## 嵌套查詢

C＃：

var nestedQueries = from s in studentList

                    where s.age > 18 && s.StandardID ==

                        (from std in standardList

                        where std.StandardName == "Standard 1"

                        select std.StandardID).FirstOrDefault()

                            select s;

nestedQueries.ToList().ForEach(s => Console.WriteLine(s.StudentName));

輸出：

Steve