LINQ (Language Integrated Query) è una tecnologia fornita da Microsoft che consente di effettuare query sui dati utilizzando un linguaggio di programmazione fortemente tipizzato, come C# o VB.NET. LINQ consente di eseguire query su vari tipi di sorgenti dati, come oggetti in memoria, database relazionali, XML, e altri.

Le query LINQ possono essere scritte in due modalità principali:

1. \*\*Sintassi Query\*\* (simile al linguaggio SQL)

2. \*\*Sintassi Lambda\*\* (funzioni anonime in forma lambda)

La sintassi lambda è molto utilizzata nelle query su database attraverso l'ORM di .NET, come \*\*Entity Framework\*\*. Di seguito ti elencherò alcune delle funzioni principali che possono essere utilizzate all'interno di lambda expression in LINQ per manipolare o leggere i dati.

### Funzioni LINQ principali utilizzabili con le lambda expression

1. \*\*`Where`\*\*

Filtra una collezione in base a una condizione espressa in lambda.

```csharp

var result = dbContext.Students.Where(s => s.Age > 18);

```

2. \*\*`Select`\*\*

Proietta i dati in una nuova forma, consente di selezionare campi specifici o creare nuovi oggetti.

```csharp

var names = dbContext.Students.Select(s => s.Name);

```

3. \*\*`OrderBy`\*\* / \*\*`OrderByDescending`\*\*

Ordina una collezione in ordine crescente o decrescente.

```csharp

var sortedStudents = dbContext.Students.OrderBy(s => s.Name);

var sortedStudentsDesc = dbContext.Students.OrderByDescending(s => s.Name);

```

4. \*\*`ThenBy`\*\* / \*\*`ThenByDescending`\*\*

Viene utilizzato dopo `OrderBy` per eseguire un ordinamento secondario.

```csharp

var sortedStudents = dbContext.Students.OrderBy(s => s.Age).ThenBy(s => s.Name);

```

5. \*\*`GroupBy`\*\*

Raggruppa gli elementi in base a una chiave.

```csharp

var groupedByAge = dbContext.Students.GroupBy(s => s.Age);

```

6. \*\*`Join`\*\*

Esegue una query di join tra due collezioni.

```csharp

var query = dbContext.Students.Join(dbContext.Courses,

student => student.CourseId,

course => course.Id,

(student, course) => new { student.Name, course.Title });

```

7. \*\*`Take`\*\* / \*\*`Skip`\*\*

`Take` restituisce un numero specifico di elementi, mentre `Skip` salta un numero specifico di elementi.

```csharp

var firstThree = dbContext.Students.Take(3);

var skipTwo = dbContext.Students.Skip(2);

```

8. \*\*`First`\*\* / \*\*`FirstOrDefault`\*\*

Restituisce il primo elemento di una collezione che soddisfa una condizione. `FirstOrDefault` restituisce `null` se non trova risultati.

```csharp

var firstStudent = dbContext.Students.First(s => s.Age > 18);

var firstOrNull = dbContext.Students.FirstOrDefault(s => s.Age > 25);

```

9. \*\*`Single`\*\* / \*\*`SingleOrDefault`\*\*

Restituisce un singolo elemento che soddisfa una condizione. Genera un'eccezione se la collezione contiene più di un elemento corrispondente. `SingleOrDefault` restituisce `null` se non trova risultati.

```csharp

var student = dbContext.Students.Single(s => s.Id == 1);

var studentOrNull = dbContext.Students.SingleOrDefault(s => s.Id == 5);

```

10. \*\*`Any`\*\*

Restituisce `true` se almeno un elemento soddisfa la condizione, altrimenti `false`.

```csharp

bool exists = dbContext.Students.Any(s => s.Age > 18);

```

11. \*\*`All`\*\*

Restituisce `true` se tutti gli elementi soddisfano una determinata condizione.

```csharp

bool allAdults = dbContext.Students.All(s => s.Age > 18);

```

12. \*\*`Count`\*\*

Restituisce il numero di elementi che soddisfano una condizione.

```csharp

int adultCount = dbContext.Students.Count(s => s.Age > 18);

```

13. \*\*`Sum`\*\*, \*\*`Min`\*\*, \*\*`Max`\*\*, \*\*`Average`\*\*

Eseguono operazioni matematiche sui campi numerici.

```csharp

int totalAge = dbContext.Students.Sum(s => s.Age);

int minAge = dbContext.Students.Min(s => s.Age);

int maxAge = dbContext.Students.Max(s => s.Age);

double averageAge = dbContext.Students.Average(s => s.Age);

```

14. \*\*`Distinct`\*\*

Restituisce solo valori unici da una collezione.

```csharp

var distinctAges = dbContext.Students.Select(s => s.Age).Distinct();

```

15. \*\*`Contains`\*\*

Verifica se un elemento esiste in una collezione.

```csharp

bool hasStudent = dbContext.Students.Select(s => s.Name).Contains("John");

```

16. \*\*`DefaultIfEmpty`\*\*

Restituisce un valore di default se la collezione è vuota.

```csharp

var students = dbContext.Students.DefaultIfEmpty();

```

17. \*\*`Aggregate`\*\*

Aggrega i valori di una collezione utilizzando una funzione.

```csharp

var concatenatedNames = dbContext.Students.Select(s => s.Name).Aggregate((current, next) => current + ", " + next);

```

18. \*\*`Zip`\*\*

Combina due collezioni in una nuova collezione di coppie.

```csharp

var zipped = dbContext.Students.Select(s => s.Name).Zip(dbContext.Students.Select(s => s.Age), (name, age) => new { Name = name, Age = age });

```

### Utilizzo di LINQ con Entity Framework

Quando si utilizza \*\*Entity Framework\*\*, LINQ viene spesso utilizzato per interrogare il database. Il vantaggio di LINQ con Entity Framework è che la query viene convertita in SQL ed eseguita direttamente sul database, ottimizzando le prestazioni e riducendo il carico sulla memoria del server applicazioni.

Ecco un esempio di come LINQ può essere utilizzato con Entity Framework:

```csharp

using (var dbContext = new SchoolContext())

{

var students = dbContext.Students

.Where(s => s.Age > 18)

.OrderBy(s => s.Name)

.Select(s => new { s.Name, s.Age })

.ToList();

}

```

### Vantaggi di LINQ

1. \*\*Fortemente tipizzato\*\*: errori di sintassi vengono catturati durante la compilazione.

2. \*\*Composizione delle query\*\*: le query LINQ possono essere composte in maniera incrementale e modificata facilmente.

3. \*\*Integrazione con il linguaggio\*\*: LINQ è parte integrante di C#, quindi non c'è bisogno di imparare una sintassi di query separata come con SQL puro.

Spero che questo ti dia una buona panoramica di LINQ e delle funzioni utilizzabili all'interno delle lambda expression!