

Capitolo 17

Introduzione a LINQ

Premetto che l'argomento oggetto di questo capitolo è non solo importante ma anche abbastanza denso di concetti e, a parer mio, non immediatamente comprensibile in tutte le sue sfumature. In questa prima stesura darò informazioni abbastanza generiche e semplici, ripromettendomi di tornarci sopra in futuro e fornire i necessari approfondimenti.

Con la sigla **LINQ** introduciamo un acronimo per Language Integrated Query. Il nome già di per se stesso è abbastanza chiaro ma in realtà dare una definizione formale precisa forse lo è un po' meno, data la grande varietà che si trova nei testi e sul Web. In accordo a quanto si trova ad esempio su Wikipedia io vedo questa tecnologia come un **framework per aggiungere nativamente ad un linguaggio capacità di querying su dati di varia natura e origine**. Anche in questo caso possiamo dire che si va nel senso dell'astrazione, ovvero è uno strato che si interpone comunque tra applicazione e dati. Anche se a qualcuno aggiungere strati non piace tanto la generalizzazione sintattica e la capacità di accesso ai dati, come vedremo, hanno decretato un buon grado di accoglienza tra gli sviluppatori. Peraltro va anche aggiunto che LINQ non nasce per eliminare SQL o risolvere tutti i problemi di chi manipola dati. Miracoli non se ne fanno.

La prima apparizione di LINQ, o meglio lo sviluppo delle sue fondamenta, risale in pratica al 2003 con il linguaggio sperimentale **Cw**, del quale esiste ancora una pagina sul sito di Microsoft Research con tanto di download del compilatore sperimentale che venne sviluppato, in cui si mossero i primi passi nello sviluppo di questa tecnologia che venne poi ufficializzata con la versione 3.0 del framework .Net. Il suo esordio nel mondo della programmazione avvenne pertanto nel 2005.

Come detto LINQ lavora sui dati e questi possono avere varie origini. Sentirete spesso parlare di "Linq to qualche cosa" e questo "qualche cosa" è il provider dei dati. In particolare nei casi predefiniti nativamente abbiamo:

- Linq to Object
- Linq to ADO.NET
- Linq to Entities
- Linq to SQL
- Linq to XML
- Linq to DataSet.

Tuttavia non è troppo difficile estendere LINQ per cui non stupitevi di vedere o sentire anche cose come Linq to LDAP, Linq to Exchange o persino Linq to Google (e perchè no?). Questo dimostra il grande eclettismo di questa estensione e pone in risalto quale sia il primo scopo di LINQ: fornire una modalità di accesso comune e coerente a dati di diversa provenienza. Questo è un aspetto che impatta fortemente sullo sviluppo in quanto l'interazione con dati di diversa provenienza è cosa del tutto comune. Va comunque detto che Linq si propone di affiancare le tecnologie esistenti, non di sostituirle. Ciascuno dei passi citati qui sopra ha obbligato i tecnici di Microsoft ad interventi piuttosto corposi.

La sintassi per eseguire le interrogazioni segue abbastanza da vicino quella SQL (Structured Query Language) il noto linguaggio di gestione dei Database, come vedremo ed è stata implementata in alcuni linguaggi basati su .Net come i noti VB e, ovviamente, C# ma anche altri ne hanno annunciato il supporto, non ho dati aggiornati a oggi (gennaio 2011) su quanti e quali siano esattamente.

Le query eseguite sono sempre soggette a tipizzazione forte il che ci garantisce un certo appoggio anche da parte del compilatore.

Come di consueto partiamo con un esempio semplice che appartiene all'ambito Linq to Object ma prima non possiamo bypassare un argomento importante, ovvero quello delle prestazioni. Sul web si trovano varie testimonianze e io stesso ho svolto, nel mio piccolo, qualche semplice prova: sembrerebbe proprio che questa tecnologia non sia "da tutti i giorni" ma vada usata con attenzione, badando a scegliere gli operatori giusti (anche se siete genietti del C# non

è detto che lo siate in ambito SQL), profilando il codice con buon senso e quando ne vale la pena in quanto introdurrebbe un overhead più che significativo in molti casi. Ad esempio il codice che segue è valido a scopo didattico ma in casi simili e con un numero molto più elevato di elementi da analizzare senza dubbio ci si deve rivolgere a quanto offre C# di serie senza tirare in mezzo Linq a tutti i costi. Mai innamorarsi di una soluzione solo perchè ci sembra più elegante e moderna di quelle classiche.

```
C# Esempio 17.1
1 using System;
2 using System.Linq;
3
4 public class Test
5 {
6     public static void Main()
7     {
8         int[] ar1 = new int[] {4, 7, 100, 2, 3, 101};
9
10        var res = from re in ar1
11                where re % 2 == 0
12                select re;
13
14        foreach (var r in res )
15            Console.WriteLine (r);
16    }
17 }
```

La riga 2 ci permette di utilizzare il necessario namespace, internamente definito nel file System.Core.dll. La 8 genera l'oggetto su cui effettueremo la query, si tratta evidentemente di un array. Le sequenze, sono il pane quotidiano di Linq. Dalla 10 alla 12 abbiamo il core del nostro esempio. E' una query in piena regola innestata nel corpo del nostro programma. Il risultato di questa una volta eseguita (in particolare quella dell'esempio 16.1 restituisce i numeri pari, ovvero quelli che ci danno resto 0 se divisi per 2) viene racchiusa in un tipo enumeratore risultato di una iterazione su di un certo insieme di valori. Per essere precisi l'istruzione

```
Console.WriteLine(res.GetType());
```

restituisce:

```
System.Linq.Enumerable+WhereArrayIterator`1[System.Int32]
```

dove la seconda parte indica un filtro basato su di un certo predicato. Manipolando appena un poco l'esempio precedente avremo un'altra risposta che forse chiarisce il concetto meglio di ogni altra spiegazione:

```
C# Esempio 17.2
1 using System;
2 using System.Linq;
3
4 public class Test
5 {
6     public static void Main()
7     {
8         string[] ar1 = new string[] {"aaa", "bbb", "abc", "bca"};
9
10        var res = from re in ar1
11                where re[0] == 'a'
12                select re;
13
14        foreach (string r in res )
15            Console.WriteLine (r);
16        Console.WriteLine(res.GetType());
17    }
18 }
```

La scrittura alla riga 16 visualizza:

```
System.Linq.Enumerable+WhereArrayIterator`1[System.String]
```

In questo caso abbiamo selezionato tramite Linq tutte le stringhe che iniziano con la lettera 'a'.

Si può dire, semplificando un po' il discorso, che gli operatori Linq, almeno finchè stiamo nel caso Linq to Object accettano in input una sequenza e restituiscono in output una sequenza modificata sulla base di quanto richiesto nella query. Le sequenze formalmente, va ricordato, devono implementare l'interfaccia `IEnumerable<T>`

In questa fase, come appare, catturiamo i dati restituiti dalla query attraverso il **tipo generico var**, di cui abbiamo già diffusamente parlato, il che è buona norma specialmente quando non si sa con che dati avremo a che fare. Avendo la certezza sul tipo di dati, come in questo caso, potremmo modificare la riga 10 come segue:

```
IEnumerable<string> res = from re in ar1
```

implementando l'interfaccia `IEnumerable` (stiamo parlando di Linq to Objects, non dimenticatelo). A disposizione delle query eseguibili tramite questa estensione del framework (avete notato che anche io cambio spesso modo di indicarlo?) abbiamo un gran numero di operatori di provenienza dal mondo SQL che la letteratura più precisa suddivide in aree legate allo scopo comune che, a gruppi, gli operatori condividono:

Scopo	Operatori SQL disponibili
Aggregazione	Aggregate, Average, Count, LongCount, Min, Max, sum
Concatenazione	Concat
Conversione	Cast, OfType, ToArray, ToDictionary, ToList, ToLookup, ToSequence
Elementi (legato agli)	DefaultEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Eguaglianza	SequenceEqual
Generazione	Empty, Range, Repeat
Di raggruppamento	GroupBy
Di unione	GroupJoin, Join
Ordinamento	OrderBy, OrderByDescending, ThenBy, ThenByAscending, Reverse
Partizionamento	Skip, SkipWhile, Take, TakeWhile
Proiezione	Select, SelectMany
Quantificatori	All, Any, Contains
Restrizione	Where
Settaggio	Distinct, Except, Intersect, Union

NB: a livello di programma C# questi operatori vanno scritti in minuscolo. In rosso sono evidenziabili quello che permettono il "differimento" di cui parleremo qui di seguito. Non è mio scopo andare ad approfondire ogni singolo comando, molti dei quali peraltro autoesplicativi, comunque sul Web sono reperibili facilmente tutte le informazioni, a partire dall'onnipresente portale MSDN. Un semplice esempio applicativo lo potete vedere all'opera con questa semplice modifica alla riga 11 dell'esempio 16.2, magari aggiungendo qualche elemento all'array che così com'è risulta un po' troppo corto:

```
where re[0] == 'a' orderby re
```

che aggiunge un ordinamento alfabetico all'output (e guardate anche cosa esce come output della riga 16, esattamente quello che ci aspettiamo anche in termini, per così dire, descrittivi).

Un cenno lo dobbiamo fare a questo punto al funzionamento interno del compilatore quando si trova a dover gestire una semplice query. Per fare questo possiamo ricorrere a tools come Ildasm o l'ottimo Reflector creato da Lutz Roeder (che merita un eterno grazie da parte di tutta la comunità .Net) ed ora gestito da RedGate. Da parte mia per ora non scenderò in grossi approfondimenti. Tuttavia vorrei evidenziare come una semplice query del tipo:

```
string[] ar1 = new string[] { "aaa", "bbb", "ccc" };
IEnumerable<string> query = from s in ar1 select s;
```

viene tradotta come

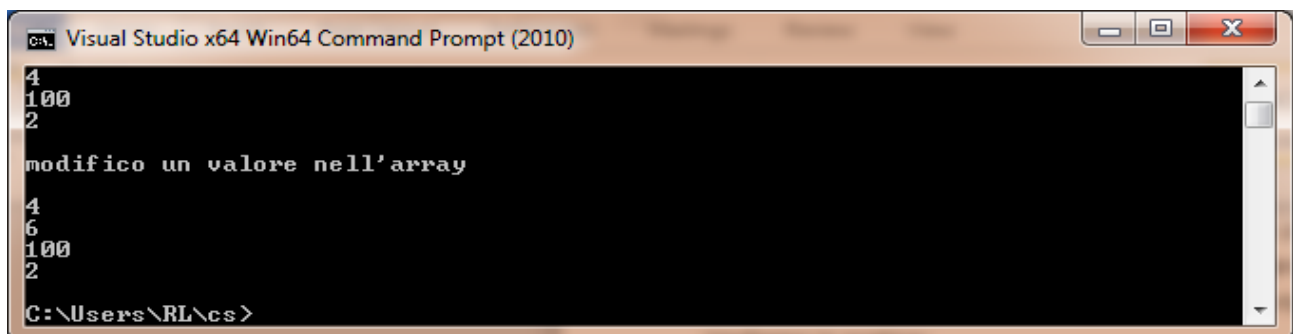
```
IEnumerable<string> query = ar1.Select<string, string>(delegate (string s)
{
return s;
})
```

in altre parole abbiamo usato un metodo esteso che prende un delegato come argomento. Ildasm invece rivelerà che a compile time viene generato un delegato anonimo. Ma, come detto, approfondirò il discorso in una versione più avanzata del capitolo. Dal breve codice presentato tuttavia comprendiamo come il compilatore internamente faccia un uso dei delegati molto più intenso di quanto facciamo noi, probabilmente. Approfondire coi tools succitati aiuta sicuramente a capirne di più del funzionamento di Linq.

Un punto a favore della nostro strumento dal punto di vista della praticità d'uso è una peculiarità che si chiama **esecuzione differita**. Questa è resa possibile da alcuni operatori (tabella precedente) ed in pratica ci garantisce che l'interrogazione sia eseguita solo al momento effettivo in cui viene richiamata garantendoci pertanto un miglior grado di aggiornamento dei dati gestiti. Anche in questo caso l'argomento è abbastanza complesso e noi, a questo stadio, vediamo solo cosa significa da un punto di vista superficiale. Se possibile cercherò di approfondire anche questo aspetto in un secondo momento. Vediamo comunque il semplice esempio seguente:

C#	Esempio 17.3
1	using System;
2	using System.Linq;
3	
4	public class Test
5	{
6	public static void Main()
7	{
8	int[] ar1 = new int[] { 4, 7, 100, 2, 3, 101};
9	var res = from re in ar1
10	where re % 2 == 0
11	select re;
12	foreach (int r in res)
13	Console.WriteLine (r);
14	Console.WriteLine("\nmodifico un valore nell'array\n");
15	ar1[1] = 6;
16	foreach (int r in res)
17	Console.WriteLine (r);
18	}
19	}

Con il seguente output:



E' evidente che la query alla riga 9-11 viene richiamata una prima volta alla riga 12 e poi, dopo la modifica di un valore nell'array, alla riga 16 dando, nel secondo caso, un risultato correttamente aggiornato. Come detto non tutti gli

operatori permettono questo tipo di differimento. La tabella precedente evidenziava in rosso quelli che lo consentono.