

INTRODUÇÃO AO LINQ

Independentemente do que alguns possam pensar ou dizer, na verdade, programar hoje em dia não é mais fácil do que há 10 ou 20 anos atrás! Apesar de, nos dias que correm, grande parte dos programadores já não se preocuparem com pormenores de baixo nível (como, por exemplo, gestão de memória), é justo reconhecermos que a complexidade associada a uma aplicação “média” é, no mínimo, elevada.

1.1 PORQUÊ O LINQ?

Basta pensarmos na diversidade de dados que alimentam uma aplicação. Por exemplo, para além das tradicionais bases de dados, hoje em dia a interacção com XML é um dado consumado. Se nos concentrarmos apenas nestas duas “origens de dados”, é fácil concluirmos que o código necessário à interacção com ambas é radicalmente diferente.

Presumimos que, se a estes pormenores técnicos juntarmos os “verdadeiros” requisitos da aplicação definidos pelo cliente, estamos todos de acordo em relação à complexidade associada à construção de uma aplicação. Felizmente para nós, a Microsoft desenvolveu uma plataforma que contribui para diminuir a complexidade inerente à pesquisa de dados oriundos de fontes de dados heterogéneas: digam olá ao LINQ! O LINQ (*Language Integrated Query*) é uma nova linguagem de pesquisa integrada na plataforma .NET cujo principal objectivo é fornecer uma nova sintaxe que permita a construção de expressões de pesquisa sobre qualquer fonte de dados. Esta nova linguagem oferece várias vantagens. Para além de permitir uma maior legibilidade, garante ainda a construção de pesquisas fortemente tipificadas sobre várias fontes de dados.

Nada como um exemplo para ilustrar algumas das potencialidades desta nova linguagem de pesquisa. Vamos supor que queremos obter uma lista de todos os processos que estão a correr na máquina, cujo nome começa por “w”.

Antes do lançamento do LINQ, teríamos de recorrer a código semelhante ao seguinte para obtermos a lista de processos desejada:

```
List<Process> antesLINQ = new List<Process>();
foreach (Process p in Process.GetProcesses()) {
    if (p.ProcessName.StartsWith("w")) {
        antesLINQ.Add(p);
    }
}
```

Com o LINQ, podemos apenas escrever o seguinte:

```
var processes = from p in Process.GetProcesses()
                where p.ProcessName.StartsWith("w")
                select p;
```

Para além da expressão LINQ, o excerto anterior introduz ainda o conceito de variável implicitamente tipificada (uma novidade do C# 3.0). A utilização do termo reservado `var` faz com que o compilador infira o tipo da variável a partir da expressão atribuída a essa variável (note-se que, no exemplo anterior, a variável `processes` é do tipo `IEnumerable<Process>`). A utilização de LINQ (neste caso, na variante de *LINQ To Objects*) tornou o nosso código mais legível, especialmente se o leitor já tiver utilizado uma linguagem declarativa (como, por exemplo, o SQL) no passado.

Se quisermos, podemos imprimir todos os elementos mantidos nessa variável através da utilização de um ciclo `foreach`:

```
foreach (var p in processes) {
    Console.WriteLine(p.ProcessName);
}
```

A Figura 1.1 ilustra os resultados obtidos após a execução do ciclo anterior.

Nesta altura, seria interessante analisarmos (ainda que superficialmente) o processamento de uma expressão LINQ.



FIGURA 1.1 – Impressão de todos os processos mantidos na variável `processes`

Quando o compilador encontra uma expressão deste tipo², encarrega-se de transformá-la em código C#. Neste caso, obteríamos o seguinte código C#:

```
var processes = Process.GetProcesses()  
    .Where(p => p.ProcessName.StartsWith("w"))  
    .Select(p => p);
```

Para já, não importa saber quais as traduções efectuadas para cada um dos termos “reservados”³ introduzidos pelo LINQ (haverá tempo para falar sobre estes aspectos mais tarde). O que importa é reter que todos os termos “reservados” LINQ são traduzidos em invocações de métodos.

Os mais atentos já deverão estar a questionar-se acerca da correcção do código anterior. Na verdade, os métodos `Where` e `Select` não foram definidos na classe `Array` (tipo devolvido pelo método `GetProcesses`), mas sim por uma nova classe estática (`static`) designada de `System.Linq.Enumerable` introduzida pela plataforma 3.5. Vejamos a assinatura do método `Where` usado no exemplo anterior:

```
public class Enumerable{
```

² O LINQ depende muito do trabalho realizado pelo compilador. Na verdade, a interpretação e tradução das expressões é da exclusiva responsabilidade do compilador.

³ Na verdade, os termos usados em expressões de pesquisa LINQ são reservados apenas no contexto associado a essa expressão. Por outras palavras, fora de uma expressão LINQ, nada nos impede de, por exemplo, definirmos uma variável designada de `from`.

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source, Func<TSource, bool>
    predicate);
}
```

O método anterior é designado de método de extensão e é outra das novidades introduzidas pela versão 3.0 da linguagem. Um método de extensão permite-nos utilizar a sintaxe de métodos de instância sobre um objecto do mesmo tipo do parâmetro prefixado com o termo `this`. Ou seja, se nos concentrarmos no método `Where` apresentado no excerto de código anterior, podemos afirmar que se `cll` for uma variável de um tipo que implemente a interface `IEnumerable<T>`, então podemos escrever `cll.Where(...)`.

É importante notar que, na prática, o método de extensão não é mais do que um método estático e que o compilador se encarrega de transformar a sintaxe de método de instância na invocação do método estático definido pela classe aquando da compilação. Por outras palavras, a invocação de métodos decorrente da execução da expressão LINQ será transformada em código C# semelhante ao seguinte:

```
var processes = Enumerable.Select(
    Enumerable.Where(Process.GetProcesses(),
        p=> p.ProcessName.StartsWith("w")),
    p => p);
```

Como seria de esperar, abordaremos os principais aspectos associados à definição e utilização de métodos de extensão neste livro (o Capítulo 9 apresenta as principais características destes tipos de métodos). Para já, importa apenas reter que a introdução dos métodos de extensão permite a invocação dos métodos `Where` e `Select` apresentados no exemplo anterior.

Existe ainda um outro aspecto importante (relativo à expressão LINQ anterior) que deve ser mencionado: ao contrário do que se possa pensar, a expressão de pesquisa só é realmente executada quando acedemos aos itens devolvidos pela pesquisa (algo que acontece, por exemplo, quando recorremos a um ciclo `foreach` para percorrer os elementos mantidos na variável `processes`). Para implementar este comportamento, são usados internamente blocos de iteração (introduzidos pela versão 2.0 do C#). Portanto, as expressões de LINQ podem ser vistas como expressões que descrevem operações que são aplicadas a dados e não como expressões que filtram e devolvem automaticamente um conjunto de dados.

Na prática, isto quer dizer que se percorrermos duas vezes os elementos mantidos na variável `processes`, poderemos, eventualmente, obter elementos diferentes em cada iteração! Como veremos, existem opções que nos permitem mudar este comportamento predefinido. Nesses cenários, a expressão de pesquisa é executada “no local onde é definida” e obtemos uma “imagem” (*snapshot*) com os dados existentes na altura da definição da expressão (note-se que para obtermos esta imagem – ou *snapshot* – temos de invocar um dos vários métodos disponibilizados pela plataforma para esse efeito – por exemplo, `ToList`. O Capítulo 10 apresenta mais informações sobre estes métodos).

As vantagens decorrentes da utilização da sintaxe LINQ seriam ainda superiores se necessitássemos, por exemplo, de agrupar os dados ou se precisássemos de efectuar projecções (aspectos estes que serão apresentados ao longo do livro).

1.2 VARIANTES DE LINQ

Apesar do exemplo anterior não o ilustrar, uma das grandes vantagens do LINQ advém do facto de podermos aplicar praticamente a mesma sintaxe para pesquisar dados provenientes de diversas fontes de dados (e, em alguns cenários, cruzá-los). Tal deve-se ao facto de podermos ter vários *providers*, cada um responsável por implementar os métodos definidos no padrão⁴ LINQ sobre uma determinada fonte de dados.

Na prática, um *provider* é sempre uma classe que disponibiliza um conjunto de métodos definidos pela especificação C# (portanto, a classe não tem de implementar nenhuma interface ou expandir uma determinada classe). Todos os *providers* apresentados neste livro não são mais do que classes estáticas que introduzem os métodos definidos no padrão LINQ através de vários métodos de extensão. A figura seguinte apresenta alguns dos *providers* apresentados ao longo deste livro.

⁴ Como veremos, o padrão LINQ mapeia cada um dos termos reservados usados no contexto de uma expressão LINQ na invocação de um método. O Capítulo 14, que descreve a criação de *providers*, apresenta estes conceitos de forma aprofundada.

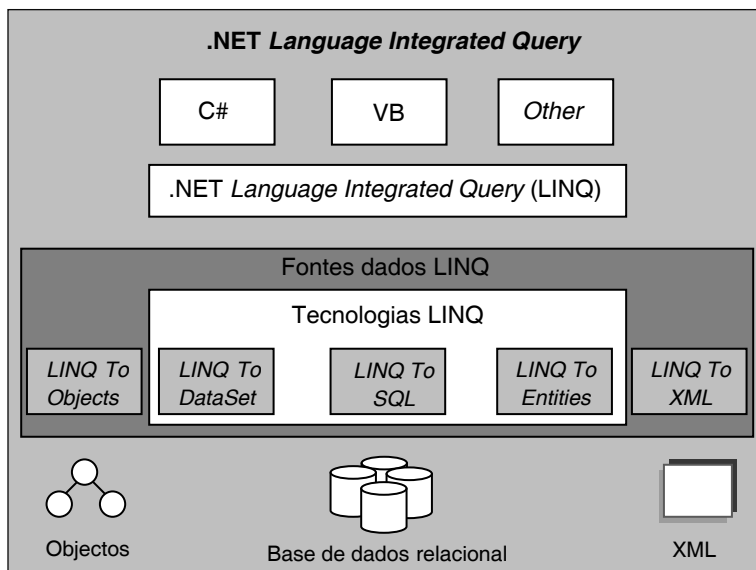


FIGURA 1.2 – Relação entre *providers* de LINQ e fontes de dados

Como é possível verificar, cada *provider* é responsável por aplicar uma expressão de dados a uma determinada fonte de dados (por exemplo, a variante *LINQ To Objects* permite-nos filtrar colecções de elementos mantidas em memória). Actualmente, um *provider* LINQ pode interagir sobre dois tipos de elementos (que representam sempre colecções): `IEnumerable<T>` e `IQueryable<T>`. A principal diferença entre os dois resume-se ao facto das expressões aplicadas sobre elementos `IQueryable<T>` serem “traduzidas” antes de serem aplicadas. Por exemplo, o *provider* *LINQ To SQL* trabalha sobre elementos do tipo `IQueryable<T>`, permitindo assim a tradução das expressões LINQ em código SQL.

Ao longo deste livro, ilustraremos a utilização de todas as variantes de LINQ apresentadas na Figura 1.2. Para além disso, veremos ainda como podemos construir os nossos próprios *providers* de LINQ.

1.3 VANTAGENS DECORRENTES DA UTILIZAÇÃO DE LINQ

Nesta altura, poderá ainda ser cedo para que o leitor já se tenha apercebido de todas as vantagens decorrentes da utilização desta tecnologia. Contudo, e como estamos perante um capítulo introdutório, optámos por

apresentar algumas das principais vantagens decorrentes da utilização desta tecnologia:

- ⊙ Unificação da sintaxe de acesso a dados — como referimos no início deste capítulo, a interacção com as diversas fontes de dados obrigava-nos, até ao momento, a conhecer os detalhes relacionados com as várias API de acesso a essas fontes. Com a introdução do LINQ, podemos recorrer a um conjunto de expressões uniformes que podem ser aplicadas a qualquer fonte de dados (claro, desde que exista um *provider* que implemente o padrão LINQ sobre essa fonte de dados!);
- ⊙ Utilização (simultânea) de vários *providers* — outra das grandes vantagens decorrentes da utilização de LINQ resulta no facto de podermos “misturar” dados provenientes de diferentes fontes de dados numa mesma expressão ou de podermos transformar facilmente um tipo de dados noutro. Por exemplo, com LINQ podemos integrar facilmente objectos mantidos em memória com outros dados provenientes de bases de dados ou de documentos XML armazenados no disco do computador;
- ⊙ Pesquisas fortemente tipificadas — para além das vantagens anteriores, o LINQ introduz uma outra vantagem: todas as expressões de pesquisa são fortemente tipificadas. Na prática, isto quer dizer que todas as expressões de LINQ são compiladas, garantindo assim que praticamente todos os erros relacionados com tipos de dados são detectados aquando dessa compilação (note-se que ainda temos de ter cuidado com eventuais erros que apenas surgem durante a execução do programa e que, por isso, não podem ser detectados aquando da compilação — por exemplo, referências nulas de objectos);
- ⊙ Fácil extensão — como vimos, as expressões LINQ dependem da existência de *providers* que são responsáveis por implementar os métodos definidos no padrão LINQ sobre uma determinada fonte de dados. Se quisermos, podemos criar um novo *provider*, responsável por interagir com dados provenientes de uma determinada fonte. A partir desta altura, podemos utilizar expressões LINQ sobre conjuntos de elementos provenientes desse tipo de fonte de dados.

Pensamos que estas vantagens deverão, nesta altura, constituir motivação suficiente para aprendizagem desta nova linguagem de pesquisa!

RESUMO

Neste primeiro capítulo, efectuámos uma apresentação muito simples da nova linguagem de pesquisa conhecida como LINQ. Para além disso, analisámos ainda algumas das principais vantagens decorrentes da utilização desta linguagem de pesquisa. Antes de analisarmos as várias variantes da linguagem, temos de iniciar o nosso percurso com a apresentação detalhada das funcionalidades disponibilizadas pela linguagem C# que servem de suporte à nova linguagem de pesquisa. Assim, no próximo capítulo vamos apresentar as principais funcionalidades associadas à definição e utilização de genéricos.