11 - Introdução à LINQ

Na anterior versão 3.0 da linguagem C#, foi introduzido um conjunto importante de funcionalidades, como inferência automática de tipos, tipos anónimos, métodos de extensão, expressões lambda e expressões de consulta. Embora cada uma dessas características seja útil por si só, na verdade, o seu poder só é realmente patente quando usadas, de forma integrada, na LINQ. A LINQ (*Language Integrated Query*) é uma "linguagem integrada de consulta" que permite tratar de uma forma uniforme dados de diferentes origens (por exemplo, bases de dados ou ficheiros XML).

A ideia da LINQ é resolver um dos maiores problemas das aplicações actuais orientadas aos objectos. Por um lado, os objectos são excelentes para serem manipulados nas linguagens de programação. Por outro lado, actualmente, o armazenamento de dados faz-se, tipicamente, em bases de dados relacionais e ficheiros XML, o que vem dificultar a sua manipulação. A LINQ consiste numa linguagem declarativa, perfeitamente integrada em C#, que minimiza a distância que existe entre dados e objectos. Consideremos o exemplo apresentado no Capítulo 7:

```
var alunoQuery =
  from   aluno in alunos
  where  aluno.Id == id
  select new { aluno.Nome, aluno.Apelido, aluno.Idade };
```

Este código irá produzir um resultado em `alunoQuery`, consistindo num conjunto de objectos, contendo o nome, apelido e idade dos alunos que têm um certo identificador. Caso a variável `alunos` represente uma tabela numa base de dados, esta é automaticamente acedida. Caso esta variável represente dados num ficheiro XML, acontecerá o mesmo. Se `alunos` representar uma simples tabela de objectos, tal como foi apresentado no exemplo original:

```
var alunos = new Aluno[]
{
  new Aluno { Id=1, Nome="Maria", Apelido="Carvalho", Idade=25 },
  new Aluno { Id=2, Nome="Pedro", Apelido="Martins",  Idade=23 },
  new Aluno { Id=3, Nome="Ana",   Apelido="Ferreira", Idade=20 },
  new Aluno { Id=4, Nome="Maria", Apelido="Cardoso",  Idade=25 },
  new Aluno { Id=5, Nome="Joao",  Apelido="Abreu",    Idade=25 }
};
```

então, apenas os dados em memória serão usados. Como se pode ver, independentemente da fonte de dados, o sistema de execução da LINQ permite tratá-los de uma forma uniforme e transparente em C#.

Em virtude da arquitectura LINQ ser demasiado abrangente, é impossível falar, de uma forma exaustiva, de todos os aspectos relacionados com esta. Na verdade, poderia escrever-se um ou mais livros apenas sobre LINQ. O leitor interessado deverá, possivelmente, adquirir um livro especificamente sobre este tema. Neste capítulo, iremos abordar de uma forma mais detalhada apenas as expressões de consulta e alguns componentes associados ao ambiente de execução (por exemplo, LINQ para SQL e LINQ para XML). Dada a relevância de algumas funcionalidades do C# para o LINQ, antes de ler este capítulo, sugerimos uma leitura rápida das seguintes secções do Capítulo 7:

- Tipos anónimos;
- Expressões de consulta;
- Inferência em expressões de consulta.

11.1 EXPRESSÕES DE CONSULTA

As expressões de consulta têm como objectivo fornecer uma forma integrada para consultas semelhantes a linguagens como o SQL ou XQuery. A Tabela 11.1 sumariza as principais palavras-chave usadas em expressões de consulta:

EXPRESSÃO	DESCRIÇÃO
`from`	Especifica uma fonte de dados e uma variável local que representa cada elemento da colecção.
`where`	Especifica critérios de restrição da consulta, seleccionando resultados que satisfaçam uma expressão lógica.
`select`	Especifica os valores que devem resultar da pesquisa.
`group`	Agrupa os resultados de uma consulta, de acordo com uma determinada chave.
`into`	Fornece um identificador que pode servir como referência aos resultados de uma cláusula `join`, `group` ou `select`.
`orderby`	Ordena, de forma ascendente ou descendente, os resultados.
`join`	Combina duas fontes de dados, usando um critério de correspondência entre eles (por exemplo, igualdade de dois campos).
`let`	Introduz uma variável local para armazenar os resultados de uma subconsulta.

Tabela 11.1 – Palavras-chave que podem ser usadas numa expressão de consulta

Nas próximas secções, iremos ver cada uma destas expressões com mais detalhe. No entanto, para o leitor que queira ver um número elevado de exemplos, sugerimos o seguinte apontador: "*Visual C# Developer Center – 101 LINQ Samples*":

http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx

Esta é uma referência essencial para os programadores de LINQ. Atrevemo-nos a dizer que, talvez, seja a forma mais prática de aprender a usar a linguagem de uma forma produtiva.

11.1.1 EXPRESSÃO FROM

Qualquer expressão de consulta começa obrigatoriamente por `from`. Esta palavra-chave especifica qual a fonte de dados envolvida na pesquisa. Obviamente, esta terá de ser iterável (por exemplo, do tipo `IEnumerable`). Considerando ainda o exemplo:

```
var alunos = new Aluno[]
{
  new Aluno { Id=1, Nome="Maria", Apelido="Carvalho", Idade=25 },
  new Aluno { Id=2, Nome="Pedro", Apelido="Martins",  Idade=23 },
  new Aluno { Id=3, Nome="Ana",   Apelido="Ferreira", Idade=20 },
  new Aluno { Id=4, Nome="Maria", Apelido="Cardoso",  Idade=25 },
  new Aluno { Id=5, Nome="Joao",  Apelido="Abreu",    Idade=25 }
};

var alunoQuery =
  from   aluno in alunos
  where  aluno.Id == id
  select new { aluno.Nome, aluno.Apelido, aluno.Idade };
```

`alunos` encontra-se nesta condição, uma vez que representa uma tabela.

É ainda possível especificar mais do que um `from` na mesma expressão de pesquisa. Por exemplo, admitindo que possuíamos uma outra tabela com as avaliações dos alunos:

```
class Avaliacao
{
  public int Id      { get; set; }   // Identificador do aluno
  public int[] Notas { get; set; }   // Notas dos testes
}
...

var avaliacoes = new Avaliacao[]
{
  new Avaliacao { Id=1, Notas=new int[] {12, 13, 14, 13} },
  new Avaliacao { Id=2, Notas=new int[] {15, 14, 16, 15} },
  new Avaliacao { Id=3, Notas=new int[] {10, 12, 14, 16} },
  new Avaliacao { Id=4, Notas=new int[] {15, 18, 17, 18} },
  new Avaliacao { Id=5, Notas=new int[] {19, 19, 18, 17} }
};
```

torna-se possível escrever uma expressão que retorna a informação do aluno, associada às suas notas. O seguinte código:

```
var alunoNotas =
  from aluno in alunos
  from notas in avaliacoes
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, aluno.Idade, notas.Notas };

foreach (var aluno in alunoNotas)
{
  Console.WriteLine("{0} {1}", aluno.Nome, aluno.Apelido);
  foreach (var nota in aluno.Notas)
    Console.Write("{0} ", nota);
  Console.WriteLine();
}
```

imprime as notas de cada aluno:

```
Maria Carvalho
12 13 14 13
Pedro Martins
15 14 16 15
Ana Ferreira
10 12 14 16
Maria Cardoso
15 18 17 18
Joao Abreu
19 19 18 17
```

Na expressão de consulta:

```
var alunoNotas =
  from aluno in alunos
  from notas in avaliacoes
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, notas.Notas };
```

indica-se, como fonte de informação, duas colecções (`alunos` e `avaliações`). Em seguida, garante-se que, ao considerar cada elemento presente em cada uma das colecções, os seus identificadores são idênticos (`aluno.Id == notas.Id`). Finalmente, emite-se um novo tipo anónimo contendo o nome, o apelido e as notas correspondentes a cada aluno.

Um ponto importante é que dentro de uma expressão `from` é possível referenciar dados de outro `from`. Por exemplo, a seguinte expressão:

```
var boasNotas =
  from aluno in alunos
  from notas in avaliacoes
  from nota in notas.Notas
  where nota > 15
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, nota };
```

permite obter uma colecção com todos os alunos que tiram notas superiores a 15 valores em alguma avaliação. Neste caso, está a tornar-se directamente visível na variável `nota` a classificação associada a cada elemento presente em `notas.Notas`. Por outro lado, `notas.Notas` representa cada elemento que terá de ser iterado em `avaliacoes`. Vale a pena estudar esta expressão com alguma atenção. O resultado da execução deste código:

```
foreach (var boas in boasNotas)
  Console.WriteLine("{0} {1}\t{2}", boas.Nome, boas.Apelido, boas.nota);
```

será

```
Pedro Martins    16
Ana Ferreira     16
Maria Cardoso    18
Maria Cardoso    17
Maria Cardoso    18
Joao Abreu       19
Joao Abreu       19
Joao Abreu       18
Joao Abreu       17
```

A RETER

LINQ – from

- Numa expressão de consulta, `from` permite especificar as fontes de dados a usar.
- `from` especifica uma fonte de dados (tipo enumerável) e uma variável que representa um elemento dessa fonte de dados. Por exemplo:

```
from aluno in alunos
```

- Numa expressão de consulta, podem utilizar-se diversas expressões com `from` simultaneamente.
- É possível numa expressão `from` utilizar um campo de uma variável introduzida noutra expressão `from`. Por exemplo:

```
from aluno in alunos
from notas in aluno.Notas
```

11.1.2 EXPRESSÃO WHERE

A expressão `where`, que temos estado a utilizar, permite filtrar o resultado, usando uma expressão lógica. Por exemplo:

```
var alunosSenior =
  from aluno in alunos
  where aluno.Idade > 23
  select aluno;
```

permite encontrar todos os alunos com mais de 23 anos. É ainda possível especificar várias expressões `where`, afectando diferentes partes da pesquisa:

```
var boasNotas =
  from aluno in alunos
  from notas in avaliacoes
  from nota in notas.Notas
  where nota > 15
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, nota };
```

ou combiná-las na mesma expressão lógica:

```
  ...
  where (aluno.Id == notas.Id) && (nota > 15)
  ...
```

Como iremos ver na secção 11.1.4, existe uma outra expressão chamada `group` que permite agrupar resultados, de acordo com um certo critério (por exemplo, calcular uma média de notas, agrupando-as por idades). A expressão `where` pode aparecer antes ou depois de uma cláusula `group`, dependendo se o objectivo é efectuar a filtragem dos elementos antes ou depois de eles serem agrupados.

Em termos de compilação, a palavra-chave `where` é convertida numa chamada ao método `Where()` (na verdade, `Enumerable.Where()`) do espaço de nomes `System.Linq`. Este método tem como objectivo efectuar filtros numa sequência de valores baseados num predicado.

A RETER

LINQ – where

- Numa expressão de consulta, `where` permite filtrar os resultados a obter de acordo com um determinado critério (expressão lógica). Por exemplo:
  ```
  where aluno.Idade>23
  ```
- Numa expressão de consulta, podem utilizar-se diversas expressões `where` simultaneamente. Alternativamente, podem combinar-se as várias expressões numa condição lógica.

11.1.3 EXPRESSÃO SELECT

Numa expressão de consulta, `select` especifica a forma e o tipo de valores que serão produzidos quando a expressão for executada. O resultado é baseado na avaliação de todas as expressões anteriores e nas expressões existentes no próprio `select`. Uma expressão de consulta tem de terminar com `select` ou `group`.

Os exemplos anteriores permitiram-nos, já, ver duas formas importantes de `select`. Em:

```
var alunosSenior =
  from aluno in alunos
  where aluno.Idade > 23
  select aluno;
```

são retornados directamente todos os elementos `aluno`, depois de devidamente filtrados. Não existe qualquer definição de novos tipos de dados. No entanto, em:

```
var alunoNotas =
  from aluno in alunos
  from notas in avaliacoes
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, notas.Notas };
```

é criado um novo tipo de dados anónimo, contendo três campos. Ambas as formas de utilização são perfeitamente lícitas. É ainda lícito emitir um tipo de dados pré-existente, utilizando os campos disponíveis na expressão de pesquisa. Por exemplo:

```
var nomesCompletos =
  from aluno in alunos
  select aluno.Nome + " " + aluno.Apelido;
```

irá criar uma lista de alunos com o seus nomes completos.

Em termos de compilação, a palavra `select` é convertida numa chamada ao método `Select()` (`Enumerable.Select()`) do espaço de nomes `System.Linq`.

A RETER

LINQ – select

- Numa expressão de consulta, `select` permite especificar quais os dados (elementos) a ser produzidos como resultado da pesquisa. Por exemplo:
  ```
  ... select aluno;
  ... select new { aluno.Nome, aluno.Idade };
  ```
- Um `select` pode emitir, tanto os dados da consulta, após filtrados, como outros tipos de dados. Neste último caso, especifica-se um tipo de dados pré-existentes a partir dos campos da consulta ou cria-se um novo tipo de dados anónimo, contendo os campos de interesse.

11.1.4 EXPRESSÃO GROUP

A expressão `group` permite agrupar elementos da pesquisa, de acordo com um certo critério. Esses elementos podem ser previamente filtrados ou não. Vejamos um exemplo simples:

```
var alunosIdade =
  from aluno in alunos
  group aluno by aluno.Idade;
```

que irá retornar uma colecção correspondente aos alunos agrupados pela idade. Cada elemento da colecção terá uma chave (a idade), contendo uma lista de alunos correspondendo a essa idade. O seguinte código:

```
foreach (var alunosPorIdade in alunosIdade)
{
  Console.WriteLine("Idade: {0} ", alunosPorIdade.Key);
  Console.WriteLine("----------------------------------");

  foreach (var aluno in alunosPorIdade)
    Console.WriteLine("  {0} {1}", aluno.Nome, aluno.Apelido);

  Console.WriteLine();
}
```

irá imprimir os alunos nesse agrupamento:

```
Idade: 25
==================================
  Maria Carvalho
  Maria Cardoso
  Joao Abreu

Idade: 23
==================================
  Pedro Martins

Idade: 20
==================================
  Ana Ferreira
```

Note-se que foi necessário utilizar dois `foreach` encadeados. O primeiro está a iterar todas as idades encontradas; o segundo está a iterar todos os alunos correspondentes a uma certa idade. Para aceder a cada uma das idades, utiliza-se a propriedade `Key`. Neste caso: `alunosPorIdade.Key`.

Vejamos mais um exemplo um pouco mais complexo. É possível especificar mais do que uma chave para realizar o agrupamento. Para tal, é necessário utilizar um tipo anónimo. O seguinte código:

```
var alunosNomeIdade =
  from aluno in alunos
  group aluno by new { aluno.Idade, aluno.Nome };

foreach (var alunosPorNomeIdade in alunosNomeIdade)
{
  Console.WriteLine("Nome: {0} ", alunosPorNomeIdade.Key);
```

```
  Console.WriteLine("==================================");
  foreach (var aluno in alunosPorNomeIdade)
    Console.WriteLine("  {0} {1}", aluno.Nome, aluno.Apelido);
  Console.WriteLine();
}
```

agrupa os alunos que têm simultaneamente a mesma idade e o mesmo primeiro nome. O resultado é:

```
Nome: { Idade = 25, Nome = Maria }
==================================
  Maria Carvalho
  Maria Cardoso

Nome: { Idade = 23, Nome = Pedro }
==================================
  Pedro Martins

Nome: { Idade = 20, Nome = Ana }
==================================
  Ana Ferreira

Nome: { Idade = 25, Nome = Joao }
==================================
  Joao Abreu
```

Em termos de compilação, a expressão `group` é convertida numa chamada ao método `GroupBy()` (`Enumerable.GroupBy()`) do espaço de nomes `System.Linq`.

A RETER

LINQ – group

- Numa expressão de consulta, `group` permite agrupar o resultado de uma consulta, usando uma certa chave. Por exemplo, agrupar todos os alunos por idade:

```
group aluno by aluno.Idade
```

- Caso se queira agrupar usando uma chave composta, é necessário definir um tipo anónimo que inclua todos os campos a introduzir na chave composta. Por exemplo:

```
group aluno by new { aluno.Idade, aluno.Nome }
```

- O resultado de uma operação de agrupamento consiste numa colecção de elementos agrupados. Para cada elemento resultado, existe uma chave correspondente ao agrupamento, acessível usando a propriedade `Key`. Para aceder aos elementos do agrupamento, é necessário iterar o elemento correspondente à chave. Por exemplo:

```
foreach (var grupoDeUmaIdade in alunosPorIdade)
{
  Console.WriteLine(grupoDeUmaIdade.Key);
  foreach (var aluno in grupoDeUmaIdade)
    Console.WriteLine(aluno);
}
```

11.1.5 EXPRESSÃO INTO

Em muitos casos, ao usar-se LINQ, é necessário encadear pesquisas ou realizar subpesquisas. A palavra-chave `into` permite definir uma variável temporária onde resultados parciais podem ser armazenados. Tal é útil quando necessitamos, mais tarde, de

utilizá-los numa outra expressão `select`, `group` ou `join` ou mesmo quando é necessário realizar filtragens intermédias.

A título de exemplo, a seguinte expressão permite agrupar os alunos por idade, filtrando, no entanto, todos os grupos que tenham menos de três pessoas:

```
var alunosIdade =
  from aluno in alunos
  group aluno by aluno.Idade into alunosNoGrupo
  where alunosNoGrupo.Count() >= 3
  select alunosNoGrupo;
```

Obviamente, é possível combinar expressões criando consultas mais complexas. No próximo exemplo, os alunos são agrupados por idade, considerando-se, apenas, os grupos que têm uma representatividade superior ou igual a 10%. Após terem sido criados os grupos, apenas se obtém o resumo de cada grupo, criando um tipo abstracto contendo a idade (chave do grupo), total de pessoas com essa idade e a percentagem correspondente:

```
var estatisticaIdades =
  from aluno in alunos
  group aluno by aluno.Idade into grupo
  where grupo.Count() >= 0.10*alunos.Count()
  select new
  {
    Idade        = grupo.Key,
    TotalPessoas = grupo.Count(),
    Percentagem  = 100.0*grupo.Count()/alunos.Count()
  };
```

Ao executar:

```
Console.WriteLine("Idade \t # \t %");
Console.WriteLine("====================");
foreach (var grupo in estatisticaIdades)
{
  Console.WriteLine("{0} \t {1} \t {2}%",
    grupo.Idade, grupo.TotalPessoas, grupo.Percentagem);
}
```

surge:

```
Idade    #       %
====================
25       3       60%
23       1       20%
20       1       20%
```

A utilização de `into` em conjunto com `group` é apenas necessária quando se quer executar operações de consulta adicionais sobre cada grupo.

A RETER

LINQ – into

- Numa expressão de consulta, `into` permite armazenar os resultados de uma consulta numa variável temporária. Consultas subsequentes podem usar essa variável como se fosse uma fonte de dados normal.
- `into` é usada quando é necessário usar dados intermédios noutra expressão `select`, `group` ou `join` ou mesmo quando é necessário realizar filtragens intermédias.

11.1.6 EXPRESSÃO ORDERBY

A palavra-chave `orderby` permite ordenar os elementos que estão a ser seleccionados numa expressão de consulta, de forma ascendente ou descendente, usando uma determinada chave de ordenamento. Por exemplo:

```
var alunosOrdenados =
  from aluno in alunos
  orderby aluno.Nome
  select aluno;
```

ordena todos os alunos por ordem ascendente de primeiro nome. Conjuntamente com `orderby`, podem utilizar-se as palavra-chave `ascending` e `descending`. Como o nome indica, estas permitem especificar a forma como é feito o ordenamento. Usando o seguinte código:

```
var alunosOrdenados =
  from aluno in alunos
  orderby aluno.Nome descending
  select aluno;
```

os alunos seriam ordenados por ordem descendente de apelido.

É também possível especificar diversas chaves de ordenação, separando-as por vírgulas (,). Por exemplo, podem ordenar-se os alunos por apelido e em seguida por nome:

```
var alunosOrdenados =
  from aluno in alunos
  orderby aluno.Apelido, aluno.Nome ascending
  select aluno;
```

Em termos de compilação, `orderby` é convertido numa chamada ao método `OrderBy()` (`Enumerable.OrderBy()`) do espaço de nomes `System.Linq`. Caso estejam presentes múltiplas chaves, estas são convertidas em chamadas ao método `ThenBy()` (`Enumerable.ThenBy()`).

A RETER

LINQ – orderby

- Numa expressão de consulta, `orderby` permite ordenar os resultados de uma pesquisa, usando uma determinada chave. Os resultados podem ser ordenados de forma ascendente (`ascending`) ou descendente (`descending`). Por exemplo:
  ```
  orderby aluno.Nota descending, aluno.Apelido ascending
  ```
- Por omissão, a expressão `orderby` ordena os elementos de forma ascendente.

11.1.7 EXPRESSÃO JOIN

Num exemplo anterior, vimos que era possível combinar informação de diferentes fontes de dados. Ao escrever-se:

```
var alunoNotas =
  from aluno in alunos
  from notas in avaliacoes
  where aluno.Id == notas.Id
  select new { aluno.Nome, aluno.Apelido, aluno.Idade, notas.Notas };
```