Технология LINQ. Выражения запросов

Выражения запросов (query expressions) — вспомогательные конструкции, позволяющие представить наиболее сложные виды запросов LINQ «в стиле SQL». Любое выражение запроса преобразуется компилятором в цепочку методов запросов; таким образом, выражения запросов — это просто конструкции языка C#, упрощающие кодирование (подобно циклу `foreach`).

В форме выражений запросов можно записать следующие виды запросов SQL:

- `Where` (группа 1),
- `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` (группа 2),
- `Select` и `SelectMany` (группа 4),
- `Join` и `GroupJoin` (группа 5),
- `GroupBy` (группа 6).

Описание грамматики выражений запросов

1) Выражение должно начинаться с конструкции `from`. Конструкция `from` определяет переменную-перечислитель для одной из последовательностей, участвующих в запросе:

```
from [тип] перечислитель in последовательность
```

В качестве последовательности может указываться не только переменная типа `IEnumerable<T>`, но и любое выражение, возвращающее значение этого типа. Конструкция `from` не имеет аналога среди методов запросов; она лишь вводит в рассмотрение последовательность и связывает с ней перечислитель. Если тип перечислителя не указан, то он определяется по типу элементов последовательности; если тип указан, то элементы последовательности приводятся к этому типу (методом `Cast`).

2) Следующая часть выражения может содержать 0 или более конструкций `join`, позволяющих организовать объединение последовательностей (методом `Join` или `GroupJoin`):

```
join [тип] внутренний_перечислитель
  in внутренная_последовательность on внешний_ключ
  equals внутренний_ключ [into идентификатор]
```

Завершающая часть конструкции (`into идентификатор`) является необязательной; при ее наличии объединение выполняется не методом `Join`, а методом `GroupJoin`. Если тип внутреннего перечислителя не указан, то он определяется по типу элементов внутренней последовательности; если тип указан, то элементы внутренней последовательности приводятся к этому типу (методом `Cast`).

3) Следующая часть выражения может содержать 0 или более конструкций `let` или `where`. Конструкция `let` вводит в запрос переменную и присваивает ей значение:

```
let идентификатор = выражение
```

Конструкция `let` не имеет аналога среди методов запросов.

Конструкция `where` обеспечивает фильтрацию элементов для последовательности или объединения последовательностей (методом `Where`):

```
where логическое_выражение
```

4) В данном месте выражения может располагаться 0 или более повторений конструкций, описанных в пунктах 1–3, начиная с конструкции `from`. С помощью нескольких конструкций `from` обеспечивается реализация метода запроса `SelectMany`.

5) Следующая часть выражения может содержать конструкцию `orderby`, состоящую из одного или более полей сортировки с необязательным указанием направления сортировки (поля разделяются запятыми):

```
orderby выражение1 [ascending | descending],
  выражение2 [ascending | descending] …
```

Конструкция аналогична вызову цепочки методов `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` (начинающейся с метода `OrderBy` или `OrderByDescending`, после которого следует 0 или более методов `ThenBy` или `ThenByDescending`).

6) Следующая часть выражения может содержать конструкцию `select` или `group`. Конструкция `select` обеспечивает проецирование исходных элементов на элементы выходной последовательности (в зависимости от количества предваряющих конструкций `from` конструкция `select` обеспечивает вызов либо метода `Select`, либо метода `SelectMany`):

```
select выражение
```

Конструкция `group` также обеспечивает проецирование исходных элементов на элементы выходной последовательности, но дополнительно выполняет группирование полученных элементов по ключу (методом `GroupBy`):

```
group выражение by выражение-ключ
```

7) В оставшейся части выражения может располагаться необязательная конструкция *продолжения запроса* `into`. Она связывает перечислитель последовательности, полученной в результате выполнения предшествующей части запроса, с идентификатором, указанным после `into`:

```
into идентификатор
```

После конструкции `into` могут располагаться другие части выражения запроса, описанные в пунктах 2–7.

Не следует смешивать конструкцию продолжения запроса `into`, с которой начинается продолжение запроса, с одноименным фрагментом, которым может завершаться конструкция `join`.

Обзор конструкций выражения запросов

Конструкции from и select

Если в конструкции `from` указывается тип перечислителя, то к последовательности применяется метод `Cast`:

```
from T e in s          | from e in s.Cast<T>()
…                      | …
```

Если после `select` указывается перечислитель из `from`, то обе конструкции заменяются на имя последовательности:

```
from e in s            | s
  select e             |
```

В любом другом случае к последовательности применяется метод `Select`, а имя перечислителя и выражение, указанное после `select`, используется в лямбда-выражении этого метода:

```
from e in s            | s.Select(e => f)
  select f             |
```

Если перед конструкцией `select` указаны две конструкции `from`, то весь этот фрагмент заменяется на вызов метода `SelectMany`, причем вторая конструкция `from` переносится в лямбда-выражение этого метода:

```
from e1 in s1          | s1.SelectMany(e1 =>
from e2 in s2          |   from e2 in s2 select f)
  select f             |
```

В случае, если после двух конструкций `from` следует конструкция, отличная от `select`, выполняется преобразование, вводящее дополнительный перечислитель `x` и вспомогательный анонимный класс (в дальнейшем полученное выражение подвергается повторному преобразованию, описанному выше):

```
from e1 in s1          | from x in
from e2 in s2          |   from e1 in s1
…                      |   from e2 in s2
                       |   select new { e1, e2 }
                       | …
```

Пример — выражение запроса для получения декартова произведения трех экземпляров одной последовательности:

```
string[] src = { "0", "1" };
var res = from e1 in src
```

```
          from e2 in src
          from e3 in src
          select e1 + e2 + e3;
foreach (var e in res)
    Console.Write(e + " ");
// 000 001 010 011 100 101 110 111
```

Выясним, в какую цепочку методов запросов будет преобразовано данное выражение. Поскольку оно начинается с двух конструкций from, за которыми *не следует* конструкция select, эти первые две конструкции from заключаются в «оболочку» вспомогательного запроса с новым перечислителем:

```
var res = from x in
              from e1 in src
              from e2 in src
              select new { e1, e2 }
          from e3 in src
          select x.e1 + x.e2 + e3;
```

Теперь перечислители e1 и e2 «погружены» в новый анонимный тип и превратились в его поля, поэтому всюду далее необходимо изменить обращения к этим перечислителям, дополнив их профиксом «x.».

Вспомогательный запрос представляет собой две конструкции from, после которых следует конструкция select. Подобные запросы заменяются на вызов метода SelectMany:

```
var res = from x in
            src.SelectMany(e1 => from e2 in src
              select new { e1, e2 })
          from e3 in src
          select x.e1 + x.e2 + e3;
```

Внешний запрос теперь также представляет собой две конструкции from, завершающиеся конструкцией select (при этом в первой конструкции from в качестве последовательность выступает возвращаемое значение метода SelectMany). Значит, его также можно преобразовать в вызов метода SelectMany:

```
var res = src.SelectMany(e1 => from e2 in src
                               select new { e1, e2 })
              .SelectMany(x => from e3 in src
                               select x.e1 + x.e2 + e3);
```

Для того чтобы полностью избавиться от элементов выражения запроса, осталось преобразовать две конструкции from … select в лямбда-выражениях:

```
var res = src.SelectMany(e1 => src.Select(e2 =>
                new { e1, e2 }))
              .SelectMany(x => src.Select(e3 =>
                x.e1 + x.e2 + e3));
```

Сравним полученную цепочку вызовов методов запросов с исходным выражением запроса:

```
var res = from e1 in src
          from e2 in src
          from e3 in src
          select e1 + e2 + e3;
```

Конструкция group

Если после group указывается перечислитель из from, то используется метод GroupBy с одним параметром:

```
from e in s                | s.GroupBy(e => k)
  group e by k             |
```

В любом другом случае используется метод GroupBy с двумя параметрами:

```
from e in s                | s.GroupBy(e => k, e => f)
  group f by k             |
```

Конструкция orderby

Приведем пример, в котором упорядочивание по всем ключам производится по возрастанию:

```
from e in s                | from e in s.OrderBy(e =>
  orderby o1, o2, …        |   o1).ThenBy(e => o2) …
```

Если после какого-либо элемента списка в конструкции orderby указывается модификатор descending, то в соответствующем месте цепочки методов запросов используется метод с суффиксом «Descending».

Конструкция let

Конструкция from, за которой следует конструкция let, преобразуется следующим образом:

```
from e in s                | from x in
  let l = v                |   from e in s
                           |   select new { e, l = v }
```

Данная конструкция позволяет в дальнейшем использовать в выражении запроса более краткий идентификатор l вместо связанного с ним выражения. Пример:

```
string[] src = { "Иван Владимиров", "Сергей Петров",
  "Виктор Сидоров", "Анатолий Васильев", "Лев Яшин" };
var res = from s in src
          let fam = s.Split()[1]
          orderby fam.Length
          select fam;
// Яшин Петров Сидоров Васильев Владимиров
```

Выясним, в какую цепочку методов запросов преобразуется данное выражение. Начнем с преобразования конструкции let:

```
var res = from x in
            from s in src
            select new {s, fam = s.Split()[1] }
          orderby x.fam.Length
          select x.fam;
```

Вложенный запрос преобразуется в метод Select:

```
var res = from x in src.Select(s =>
                       new {s, fam = s.Split()[1] })
          orderby x.fam.Length
          select x.fam;
```

Теперь можно преобразовать конструкцию orderby:

```
var res = from x in src.Select(s =>
                       new {s, fam = s.Split()[1] })
                      .OrderBy(x => x.fam.Length)
          select x.fam;
```

Наконец, преобразуем конструкцию select в метод Select:

```
var res = src.Select(s =>
                new {s, fam = s.Split()[1] })
             .OrderBy(x => x.fam.Length)
             .Select(x => x.fam);
```

Сравним полученную цепочку методов с исходным выражением запроса:

```
var res = from s in src
          let fam = s.Split()[1]
          orderby fam.Length
          select fam;
```

Если попытаться реализовать требуемый запрос «на пустом месте» (не выводя его из выражения запроса), то полученная цепочка методов окажется короче выражения запроса, не уступая ему в наглядности:

```
var res = src.Select(s => s.Split()[1])
             .OrderBy(x => x.Length);
```

Конструкция where

Конструкция from, за которой следует конструкция where, преобразуется следующим образом:

```
from e in s                  | from e in s.Where(e => w)
  where w                    |
```

Конструкция join

Конструкция join, вместе с предшествующей ей конструкцией from и последующей конструкцией select преобразуется двумя различными способами, в зависимости от наличия или отсутствия элемента into:

```
from e1 in s1                | from x in s1.Join(s2,
  join e2 in s2              |   e1 => k1, e2 => k2,
  on k1 equals k2            |   (e1, e2) => f)
  select f                   | select x
-----------------------------+-----------------------------
from e1 in s1                | from x in s1.GroupJoin(s2,
  join e2 in s2              |   e1 => k1, e2 => k2,
  on k1 equals k2            |   (e1, i) => f)
  into i                     | select x
  select f                   |
```

Элемент into i является не просто формальным признаком того, что следует использовать метод GroupJoin вместо Join. Определяемый в этом элементе идентификатор i обозначает *последовательность* тех элементов последовательности s2, которые имеют тот же ключ, что и элемент e1 последовательности s1. Именно идентификатор i входит в качестве второго параметра в последнее лямбда-выражение метода GroupJoin. Таким образом, в первом варианте конструкции join (без элемента into) в выражении f можно использовать перечислители e1 и e2, тогда как во втором варианте этой конструкции (с элементом into) в выражении f следует использовать перечислитель e1 и последовательность i. В обоих вариантах при определении ключа k1 надо использовать перечислитель e1, а при определении ключа k2 — перечислитель e2.

Если конструкция join не предшествует непосредственно конструкции select, то подобный фрагмент преобразуется во вложенный запрос, содержащий дополнительный перечислитель и вспомогательный анонимный класс:

```
from e1 in s1                | from x in
  join e2 in s2              |   from e1 in s1
  on k1 equals k2            |   join e2 in s2
…                            |   on k1 equals k2
                             |   select new { e1, e2 }
                             | …
-----------------------------+-----------------------------
from e1 in s1                | from x in
  join e2 in s2              |   from e1 in s1
  on k1 equals k2            |   join e2 in s2
  into i                     |   on k1 equals k2
…                            |   into i
                             |   select new { e1, i }
                             | …
```

Если в конструкции join указывается тип перечислителя e2, то к последовательности s2 применяется метод Cast:

```
join T e2 in s2              | join e2 in s2.Cast<T>()
…                            | …
```

В качестве примера реализуем в виде выражений запросов все три запроса, рассмотренных при описании методов Join и GroupJoin (группа 5):

```
int[] src1 = { 10, 21, 33, 84 };
int[] src2 = { 40, 51, 52, 53, 60 };
var res1 = src1.Join(src2, a => a % 10, b => b % 10,
  (a, b) => a + "-" + b);
var res2 = src1.GroupJoin(src2, a => a % 10,
  b => b % 10, (a, bb) => a + "-" + bb.Count());
var res3 = src1.GroupJoin(src2, a => a % 10,
  b => b % 10,  (a, bb) =>
  bb.DefaultIfEmpty().Select(e => a + "-" + e))
  .SelectMany(e => e);
```

Во всех запросах ключом является последняя (правая) цифра числа. В первом запросе строится плоское внутреннее объединение, во втором — иерархическое левое внешнее объединение и, наконец, в третьем — плоское левое внешнее объединение.

```
res1: 10-40, 10-60, 21-51, 33-53
res2: 10-2, 21-1, 33-1, 84-0
res3: 10-40, 10-60, 21-51, 33-53, 84-0
```

Те же запросы, оформленные в виде выражений запросов:

```
int[] src1 = { 10, 21, 33, 84 };
int[] src2 = { 40, 51, 52, 53, 60 };
var res1 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           select a + "-" + b;
var res2 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           into bb
           select a + "-" + bb.Count();
var res3 = from a in src1
           join b in src2
           on a % 10 equals b % 10
           into bb
           from e in bb.DefaultIfEmpty()
           select a + "-" + e;
```

Рассмотрим, как будет выполняться преобразование последнего запроса. В нем конструкция join не завершается конструкцией select, поэтому данная конструкция преобразуется во вложенный запрос, связанный с дополнительным перечислителем x:

```
var res3 = from x in
             from a in src1
             join b in src2
             on a % 10 equals b % 10
             into bb
             select new { a, bb }
           from e in x.bb.DefaultIfEmpty()
           select x.a + "-" + e;
```

Данное выражение запроса содержит две «внешние» конструкции from (from x и from e), завершающиеся конструкцией select, поэтому оно преобразуется к методу SelectMany, возвращающему «плоскую» последовательность элементов.

Конструкция into

Конструкция into i (i — идентификатор), с которой начинается продолжение запроса, преобразуется следующим образом:

```
предшествующий               | from i in
  запрос                     |   предшествующий запрос
into i                       | продолжение запроса
продолжение запроса          |
```

Таким образом, конструкция into позволяет размещать фрагменты сложных запросов в последовательную цепочку, не оформляя начальную часть в виде вложенного запроса.