Технология LINQ. Запросы LINQ to Objects

Технология LINQ (Language Integrated Query — «запрос, интегрированный в язык») обеспечивает «встраивание» в язык программирования возможности использования запросов, подобных запросам SQL. LINQ — это набор функциональных возможностей языка C# и платформы .NET, обеспечивающих написание безопасных в смысле типизации запросов к локальным коллекциям объектов и удаленным источникам данных.

Компоненты LINQ API версии .NET Framework 3.5:

- **LINQ to Objects** — базовый интерфейс для стандартных запросов к локальным коллекциям; основан на методах класса `System.Linq.Enumerable` (ок. 40 видов запросов);
- **LINQ to XML** — интерфейс, предназначенный для обработки XML-документов; включает не только набор дополнительных запросов (ок. 10 видов запросов), но и новую *объектную модель* XML-документов, представленную иерархией классов из пространства имен `System.Xml.Linq`;
- **LINQ to SQL** и **LINQ to Entities** — интерфейсы, предназначенные для взаимодействия с базами данных Microsoft SQL Server в качестве источников удаленных наборов данных, основаны на методах класса `System.Linq.Queryable`.

Для возможности применения к коллекции запросов LINQ необходимо, чтобы коллекция реализовывала *обобщенный* интерфейс `IEnumerable<T>`. «Старые» коллекции из пространства имен `System.Collections` можно преобразовать в типу `IEnumerable<T>` с помощью *запросов импортирования* `OfType` и `Cast`. Все классы, реализующие интерфейс `IEnumerable<T>`, будем называть *последовательностями*.

Технология LINQ и новые возможности языка C# 3.0

Большинство нововведений языка C# версии 3.0 связано именно с технологией LINQ и призвано максимально упростить код, связанный с запросами LINQ. Прежде всего, следует указать лямбда-выражения, заменяющие анонимных делегатов.

Пусть в программе дан набор фамилий, из которого надо отобрать фамилии, длина которых превосходит 6 символов:

```
string[] data = { "Владимиров", "Петров", "Сидоров",
  "Васильев", "Яшин" };
IEnumerable<string> res = Enumerable.Where(data,
  delegate(string s){ return s.Length > 6; });
```

Вариант с использованием лямбда-выражений:

```
IEnumerable<string> res = Enumerable.Where(data,
  s => s.Length > 6);
```

Еще короче: используем ключевое слово `var`:

```
var res = Enumerable.Where(data, s => s.Length > 6);
```

Вывод полученной последовательности — циклом `foreach`:

```
foreach (string s in res)
  Console.Write(s + " ");
```

Результат:

```
Владимиров Сидоров Васильев
```

Пусть требуется дополнительно отсортировать полученную последовательность по алфавиту (в лексикографическом порядке):

```
var res = Enumerable.Where(data, s => s.Length > 6);
res = Enumerable.OrderBy(res, s => s);
```

В результате элементы полученной последовательности будут отсортированы по алфавиту:

```
Васильев Владимиров Сидоров
```

Вариант — один оператор с вложенными запросами:

```
var res = Enumerable.OrderBy(Enumerable.Where(data,
  s => s.Length > 6), s => s);
```

Этот вариант не обладает достаточной наглядностью: (1) многократно используется имя `Enumerable`, (2) порядок *записи* запросов не совпадает с порядком их *вызова*.

Для повышения наглядности надо использовать *методы расширения* (extensions). Все методы класса `Enumerable` являются статическими методами расширения, поэтому их можно вызывать, указывая в качестве вызывающего класса саму исходную последовательность:

```
var res = data.Where(s => s.Length > 6);
res = res.OrderBy(s => s);
```

Последовательное выполнение запросов принимает вид *цепочки* запросов:

```
var res=data.Where(s => s.Length>6).OrderBy(s => s);
```

Все элементы цепочки (кроме, быть может, последнего) должны представлять запросы, возвращающие последовательности. Еще пример::

```
data.Where(s => s.Length > 6).Count(); // 3
```

Важную роль в применении технологии LINQ играют также *анонимные типы* — еще одно нововведение C# 3.0. Они обычно используются при преобразовании исходной последовательности в последовательность элементов другого типа (так называемое *проецирование* последовательности). Пример:

```
string[] src = { "Иван Владимиров", "Сергей Петров",
"Виктор Сидоров", "Анатолий Васильев", "Лев Яшин" };
```

Получим проекцию данной последовательности в последовательность элементов с двумя полями: `FirstName` и `LastName`:

```
var res = src.Select(s => { string[] ss = s.Split();
    return new { FirstName = ss[0],
      LastName = ss[1] }; })
  .Where(s => s.LastName.Length > 6)
  .OrderBy(s => s.LastName);
foreach (var s in res)
  Console.WriteLine(s);
```

Результат:

```
{ FirstName = Анатолий, LastName = Васильев }
{ FirstName = Иван, LastName = Владимиров }
{ FirstName = Виктор, LastName = Сидоров }
```

В данном случае использование ключевого слова `var` является *обязательным* как при описании возвращаемой последовательности `res`, так и при описании параметра `s` цикла `foreach`.

В данном примере лямбда-выражение метода `Select` содержит не возвращаемое значение (в виде выражения), а *операторный блок*. Для сложных цепочек запросов целесообразно отображать их на нескольких строках, начиная каждую строку с вызова очередного запроса, предваренного точкой.

Особенности технологии LINQ

Запросы LINQ *никогда не изменяют входную последовательность*: если запрос предназначен для преобразования входной последовательности, то преобразованная последовательность возвращается как результат выполнения запроса. Такое поведение соответствует парадигме *функционального программирования*, на которой основана технология LINQ.

Запросы LINQ стараются сохранить исходный порядок элементов при преобразовании входной последовательности в выходную. В частности, порядок элементов сохраняется при выполнении запросов `Where` и `Select`; более того, в лямбда-выражениях для данных запросов предусмотрена возможность использования *индекса* элемента входной последовательности. Во всех случаях, когда в результате выполнения запроса отбрасываются повторяющиеся элементы, в результирующей последовательности остается *первое вхождение* каждого повторяющегося элемента. К числу немногих запросов, явным образом изменяющих порядок следования элементов, относится запрос инвертирования `Reverse` и все виды запросов сортировки.

Важной особенностью запросов, возвращающих последовательность, является их *отложенный* (или «*ленивый*») характер: все подобные запросы выполняются не в момент конструирования (и присваивания некоторому объекту типу `IEnumer-`

able<T>), а при *переборе* элементов полученной последовательности (обычно этот перебор реализуется посредством цикла foreach). Это означает, что если между конструированием отложенного запроса и перебором выходной последовательности будут внесены изменения во входную последовательность, то эти изменения будут учтены при переборе результата. Далее, это означает, что отложенный запрос будет *выполнен повторно*, если будет организован повторный перебор результирующей последовательности. Наконец, если при конструировании запроса в его лямбда-выражениях используются *внешние переменные*, то будет приниматься во внимание значение этих переменных не в момент *конструирования* запроса, а в момент *перебора* выходной последовательности. Пример:

```
int[] src = { 12, 14, 60, 32, 48, 70 };
var res = src.AsEnumerable();
int k = 0;
res = res.Where(n => n % 10 != k);
k = 2;
res = res.Where(n => n % 10 != k);
foreach (var e in res)
  Console.WriteLine(e); // 14 60 48 70
```

При выполнении сконструированного запроса из исходной последовательности будут удалены только числа, оканчивающиеся на 2 (а оканчивающиеся на 0 останутся), поскольку в момент *выполнения* запроса (который наступит при начале цикла foreach) значение переменной k будет равно 2, и это значение будет использовано во *всех* лямбда-выражениях, входящих в сконструированный запрос.

Сконструированный запрос выполняется немедленно, если его результатом является скалярное значение или коллекция определенного типа (например, массив).

Обзор запросов LINQ to Objects

Для быстрого ознакомления с доступными запросами и их параметрами удобно использовать технологию IntelliSense: достаточно набрать после имени последовательности точку и выбрать из появившегося списка требуемый метод расширения (после имен методов расширения указываются угловые квадратные скобки, например, «Aggregate <>»). После выбора имени метода расширения и ввода скобки «(» во всплывающей подсказке выводится список параметров метода или варианты списка параметров, если метод является перегруженным.

Соглашения, используемые в описаниях запросов:

- после названия группы запросов вначале указывается тип объекта, к которому должны применяться запросы этой группы (почти всегда это IEnumerable<TSource>; исключение составляют группы 2, 5, 7 и 12), а затем (после символа «→») — тип возвращаемого значения для запросов этой группы (если запросы группы могут возвращать значения различных типов, то тип возвращаемого значения, предваряемый символом «→», указывается после описания каждого запроса — см. группы 6, 8, 10, 12);
- элементы описания запроса, которые могут быть опущены, заключаются в квадратные скобки [ ];
- дополнительные параметры типа IComparer<T> и IEqualityComparer<T>, присутствующие в реализации некоторых перегруженных методов (см. группы 2, 5, 6, 8, 11), но используемые достаточно редко, не указываются в списке параметров;
- если параметр запроса представляет собой делегат (это всегда обобщенный делегат из семейства Func), то указывается не имя этого делегата, а соответствующее *лямбда-выражение*, вместе с типами входных параметров и типом возвращаемого значения (например, вместо предиката Func<TSource, [int,] bool> predicate указывается лямбда-выражение (TSource[, int]) => bool).

Группа 1. Фильтрация, инвертирование и преобразование пустой последовательности

```
IEnumerable<TSource> → IEnumerable<TSource>
Where((TSource[, int]) => bool)
TakeWhile((TSource[, int]) => bool)
SkipWhile((TSource[, int]) => bool)
Take(int count)
Skip(int count)
Distinct()
Reverse()
DefaultIfEmpty([TSource defaultValue])
```

Метод Where возвращает те элементы входной последовательности (в том же порядке), для которых указанный предикат возвращает значение true; метод TakeWhile заносит в выходную последовательность элементы входной последовательности, пока указанный предикат возвращает значение true; метод SkipWhile пропускает начальные элементы входной последовательности, пока предикат возвращает значение true, после чего заносит в выходную последовательность все оставшиеся элементы. Во всех трех методах предикат может содержать дополнительный параметр — индекс анализируемого элемента.

Методы Take и Skip, подобно методам TakeWhile и SkipWhile, возвращают начальную или, соответственно, конечную часть исходной последовательности, однако для этих методов явно указывается число начальных элементов, которые надо возвратить (метод Take) или пропустить (метод Skip).

Метод Distinct возвращает последовательность без повторяющихся элементов (в последовательности оставляются только первые вхождения повторяющихся элементов).

Метод Reverse возвращает последовательность, в которой порядок следования элементов изменен на обратный.

Метод DefaultIfEmpty возвращает непустую входную последовательность в неизменном виде, а в случае пустой входной последовательности возвращает последовательность, содержащую *единственный* элемент, значение которого равно defaultValue, если этот параметр указан, или default(TSource), если метод вызван без параметра (default(TSource) означает null для ссылочных типов и структуру с побитово обнуленными полями для размерных типов). Пример использования метода DefaultIfEmpty будет приведен далее (см. замечание в конце описания группы 5).

Группа 2. Упорядочивание

```
IEnumerable<TSource> → IOrderedEnumerable<TSource>
OrderBy(TSource => TKey)
OrderByDescending(TSource => TKey)
ThenBy(TSource => TKey)
ThenByDescending(TSource => TKey)
```

Методы возвращают элементы исходной последовательности, отсортированные по указанному *ключу*. Ключ определяется лямбда-выражением и должен иметь тип, реализующий интерфейс IComparable. Если тип ключа не реализует указанный интерфейс (или если при сортировке надо использовать способ упорядочивания, отличный от стандартного), можно использовать перегруженный вариант любого из приведенных методов, содержащий дополнительный второй параметр comparer типа IComparer<TKey>.

Методы OrderBy и OrderByDescending выполняют *неустойчивую* сортировку (исходный порядок элементов с одинаковыми ключами в результате сортировки может измениться).

Методы ThenBy и ThenByDescending используются, если последовательность требуется отсортировать по *набору ключей*; эти методы переупорядочивают (в соответствии со своим ключом) только те элементы последовательности, у которых были

одинаковые ключи на предыдущем этапе сортировки. Методы ThenBy и ThenByDescending могут вызываться только для уже отсортированных последовательностей (типа IOrderedEnumerable<TSource>), поэтому первым в цепочке сортирующих методов должен быть либо OrderBy, либо OrderByDescending.

Тип IOrderedEnumerable<T> может неявно приводиться к типу IEnumerable<T>, поэтому к отсортированной последовательности можно применять любые другие операции запросов, однако при этом преобразованные последовательности уже будут иметь «неотсортированный» тип IEnumerable<T>.

Группа 3. Сцепление и теоретико-множественные операции

IEnumerable<TSource> → IEnumerable<TSource>

```
Concat(IEnumerable<TSource> second)
Union(IEnumerable<TSource> second)
Intersect(IEnumerable<TSource> second)
Except(IEnumerable<TSource> second)
```

Метод Concat возвращает последовательность, содержащую все элементы первой входной последовательности (для которой вызван данный метод), после которых следуют все элементы второй последовательности (параметра second).

Остальные методы реализуют теоретико-множественные операции (объединение, пересечение и разность) для двух исходных последовательностей. Последовательность, полученная в результате выполнения любого из этих методов, не содержит повторяющихся элементов. Порядок следования элементов определяется порядком их первых вхождений в первую исходную последовательность.

Группа 4. Проецирование

IEnumerable<TSource> → IEnumerable<TResult>

```
Select((TSource[, int]) => TResult)
SelectMany((TSource[, int]) => IEnumerable<TResult>)
```

В методе Select число элементов во входной и выходной последовательностях одинаково, но их тип может различаться.

В методе SelectMany элемент входной последовательности может быть преобразован в *несколько* (0 или более) элементов выходной последовательности. Метод позволяет преобразовать иерархическую последовательность (последовательность последовательностей) в «плоскую» последовательность. Пример:

```
string[] src = { "AB CD", "E F G", "XYZ" };
var res = src.SelectMany(s => s.Split());
// AB, CD, E, F, G, XYZ
```

Если бы вместо метода SelectMany был вызван Select, то последовательность res состояла бы из трех элементов – *строковых массивов* и имела бы тип IEnumerable<string[]>.

Если бы в методе SelectMany лямбда-выражение имело вид s => s, то была бы получена последовательность *символов*, содержащихся в элементах массива src (класс string реализует интерфейс IEnumerable<char> и поэтому может рассматриваться как *символьная последовательность*).

С помощью комбинации методов Select и SelectMany можно организовать перебор *упорядоченных пар* элементов двух последовательностей (их *декартово произведение*) и вернуть результат — «плоскую» последовательность. Пример:

```
int[] src1 = { 10, 20, 30 }, src2 = { 0, 1, 2, 3 };
var res = src1.SelectMany(a => src2.Select(b => a+b));
// 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33
```

Группа 5. Объединение

IEnumerable<TOuter> → IEnumerable<TResult>

```
Join(IEnumerable<TInner> inner, TOuter => TKey,
  TInner => TKey, (TOuter, TInner) => TResult)
GroupJoin(IEnumerable<TInner> inner, TOuter => TKey,
  TInner => TKey, (TOuter,
  IEnumerable<TInner>) => TResult)
```

В методах Join и GroupJoin, как и в методах группы 3, используются две исходных последовательности: первая («внешняя») последовательность вызывает методы, а вторая («внутренняя») указывается в качестве первого параметра inner.

Метод Join выполняет *внутреннее объединение* двух последовательностей по ключу. Ключи определяются первыми двумя лямбда-выражениями; к каждой паре элементов внешней и внутренней последовательности, имеющих одинаковые ключи, применяется третье лямбда-выражение, и его результат заносится в выходную последовательность. Объединение является *внутренним*: учитываются только те элементы внешней последовательности, для которых найден *хотя бы один* элемент внутренней последовательности с таким же ключом.

Метод GroupJoin тоже выполняет объединение двух последовательностей по ключу, однако результирующий элемент определяется по элементу внешней последовательности и *всем* элементам внутренней последовательности с тем же ключом. Объединение является *левым внешним*; в нем *любой* элемент из первой («левой») последовательности примет участие в формировании выходной последовательности, даже если для него не найдется «парных» элементов из второй последовательности.

```
int[] src1 = { 10, 21, 33, 84 };
int[] src2 = { 40, 51, 52, 53, 60 };
var res = src1.Join(src2, a => a % 10, b => b % 10,
  (a, b) => a + "-" + b);
// 10-40, 10-60, 21-51, 33-53
```

При использовании метода GroupJoin надо изменить последнее лямбда-выражение (метод Count описан в группе 10):

```
var res = src1.GroupJoin(src2, a => a % 10,
  b => b % 10, (a, bb) => a + "-" + bb.Count());
// 10-2, 21-1, 33-1, 84-0
```

Для данных методов предусмотрен вариант с дополнительным параметром comparer типа IEqualityComparer<TKey>, позволяющим переопределить способ сравнения ключей.

Для методов Join и GroupJoin используется эффективная реализация, не сводящаяся к двойному циклу с попарными проверками ключей: внутренняя последовательность предварительно преобразуется к индексированной по ключу *таблице просмотра* типа ILookup (см. группу 8), что позволяет находить ее элементы, парные к элементам внешней последовательности, не прибегая к их многократному перебору.

Построение плоского внешнего объединения. Метод Join возвращает «плоское» внутреннее объединение, а метод GroupJoin — иерархическое левое внешнее объединение. Чтобы получить плоское внешнее объединение, необходимо применить цепочку из запросов GroupJoin и SelectMany. Однако простое применение к результату запроса GroupJoin метода SelectMany вернет *внутреннее* объединение:

```
var res = src1.GroupJoin(src2, a => a % 10, b =>
  b % 10, (a, bb) => bb.Select(e => a + "-" + e))
  .SelectMany(e => e);
// 10-40, 10-60, 21-51, 33-53
```

Причина: метод SelectMany *игнорирует все пустые последовательности*. Проблема решается с помощью метода DefaultIfEmpty, позволяющего преобразовать пустую последовательность в одноэлементную последовательность (см. группу 1):

```
var res = src1.GroupJoin(src2, a => a % 10, b =>
  b % 10, (a, bb) =>
  bb.DefaultIfEmpty().Select(e => a + "-" + e))
  .SelectMany(e => e);
// 10-40, 10-60, 21-51, 33-53 84-0
```

Получено плоское левое внешнее объединение.

Группа 6. Группирование

```
IEnumerable<TSource> → последовательность другого типа
GroupBy(TSource => TKey)
  → IEnumerable<IGrouping<TKey, TSource>>
GroupBy(TSource => TKey, TSource => TElement)
  → IEnumerable<IGrouping<TKey, TElement>>
GroupBy(TSource => TKey, (TKey,
  IEnumerable<TSource>) => TResult)
  → IEnumerable<TResult>
GroupBy(TSource => TKey, TSource => TElement,
  (TKey, IEnumerable<TElement>) => TResult)
  → IEnumerable<TResult>
```

Метод GroupBy реорганизует элементы «плоской» входной последовательности в последовательность *групп* элементов (т. е. в иерархическую последовательность). Является в некоторым смысле «обратным» к методу SelectMany.

Группирование производится по *ключу*; лямбда-выражение для ключа является первым параметром. Если этот параметр единственный, то выходная последовательность содержит элементы типа IGrouping<TKey, TSource>, которые сами являются подпоследовательностями с элементами типа TSource и дополнительно содержат свойство для чтения Key — ключ, связанный с данной подпоследовательностью:

```
public interface IGrouping<TKey, TElement>
  : IEnumerable<TElement>, IEnumerable
{ TKey Key { get; } }
```

Если в качестве второго параметра метода GroupBy указать лямбда-выражение вида TSource => TElement то в подпоследовательность будут включаться объекты типа TElement.

Элементы результирующей последовательности можно определить явно; для этого надо использовать лямбда-выражение, которое по ключу и связанной с ним подпоследовательности определяет элемент выходной последовательности.

```
int[] src = { 10, 21, 33, 84, 40, 51, 52, 53, 60 };
var res = src.GroupBy(i => i % 10);
foreach (var g in res) // g имеет тип
{                      // IGrouping<int, int>
    Console.Write("Key = " + g.Key + ":");
    foreach (var n in g)
        Console.Write(" " + n);
    Console.WriteLine();
}
// Key = 0: 10 40 60
   Key = 1: 21 51
   Key = 3: 33 53
   Key = 4: 84
   Key = 2: 52
```

Пример для второго перегруженного варианта:

```
var res = src.GroupBy(i => i % 10, i => i / 10);
// Key = 0: 1 4 6
   Key = 1: 2 5
   Key = 3: 3 5
   Key = 4: 8
   Key = 2: 5
```

Можно явно определить выходную последовательность, используя анонимный тип:

```
var res = src.GroupBy(i => i % 10,
  (k, nn) => new { Key = k, Sum = nn.Sum() });
foreach (var g in res)
  Console.WriteLine("Key = " + g.Key + ": " + g.Sum);
```

Мы воспользовались методом Sum (см. группу 10). Результат:

```
// Key = 0: 110
   Key = 1: 72
   Key = 3: 86
   Key = 4: 84
   Key = 2: 52
```

Пример для последнего варианта (свяжем с каждым ключом сумму чисел, в которых отброшена последняя цифра):

```
var res = src.GroupBy(i => i % 10, i => i / 10,
  (k, nn) => new { Key = k, Sum = nn.Sum() });
// Key = 0: 11
   Key = 1: 7
   Key = 3: 8
   Key = 4: 8
   Key = 2: 5
```

Имеются перегруженные варианты с дополнительным параметром comparer типа IequalityComparer<TKey>, позволяющим переопределить способ сравнения ключей.

Группа 7. Импортирование

```
IEnumerable → IEnumerable<TResult>
OfType<TResult>()
Cast<TResult>()
```

Методы предназначены, прежде всего, для преобразования коллекций, реализующих только необобщенный интерфейс IEnumerable (например, ArrayList), в последовательности IEnumerable<T>. Они различаются способом обработки особой ситуации: если тип элемента входной коллекции отличен от TResult и, кроме того, тип TResult не входит в число его предков, то OfType *игнорирует* элемент, не включая его в выходную последовательность, а Cast *возбуждает исключение*.

С помощью метода OfType нельзя выполнять преобразования скалярных типов (например, int в long, long в int, int в char и т. п.), поскольку участвующие в этих преобразованиях типы не связаны отношениями «предок–потомок». Пример:

```
int[] src = {1, 2, 3};
var res = src.OfType<long>();
// res не содержит ни одного элемента
```

Метод OfType будет выполнять преобразование элемента e к типу TResult только если операция e is TResult вернет значение true. Для преобразования, например, последовательности типа IEnumerable<int> в последовательность типа IEnumerable<long> можно воспользоваться методом Select:

```
var res = src.Select(n => (long)n);
```

В данном случае на этапе компиляции известно, что n имеет тип int, и поэтому компилятор может выполнить преобразование типа int к типу long.

Метод OfType может также применяться для *отбора* из исходной коллекции элементов-потомков определенного типа. Например, если у класса T1 имеются два класса-потомка T2 и T3, то для отбора из коллекции src типа IEnumerable<T1> только тех элементов, фактический тип которых равен T2, достаточно выполнить следующий оператор:

```
var res1 = src.TypeOf<T2>();
```

Подобного результата можно добиться и методом Where:

```
var res2 = src.Where(e => e is T2);
```

Последовательности res1 и res2 будут содержать одни и те же элементы, однако их *тип* будет различным: последовательность res1 имеет тип IEnumerable<T2>, в то время как последовательность res2 — «старый» тип IEnumerable<T1>. Таким образом, чтобы получить последовательность, полностью идентичную res1, не применяя метод TypeOf, придется воспользоваться менее эффективной цепочкой из двух запросов:

```
var res3 = src.Where(e => e is T2).Select(e => (T2)e);
```

Группа 8. Экспортирование

```
IEnumerable<TSource> → коллекция определенного типа
ToArray() → TSource[]
ToList() → List<TSource>
ToDictionary(TSource => TKey)
  → Dictionary<TKey, TSource>
ToDictionary(TSource => TKey, TSource => TElement)
  → Dictionary<TKey, TElement>
ToLookup(TSource => TKey) → ILookup<TKey, TSource>
ToLookup(TSource => TKey, TSource => TElement)
  → ILookup<TKey, TElement>
AsEnumerable<TSource>() → IEnumerable<TSource>
AsQueryable<TSource>() → IQueryable<TSource>
```

Методы ToArray, ToList, ToDictionary преобразуют входную последовательность в коллекцию указанного типа; при этом сконструированный запрос выполняется *немедленно*. При выполнении метода ToDictionary обязательно указывается лямбда-выражение для вычисления *ключа* словаря по элементу входной последовательности; при этом необходимо, чтобы для каждого элемента входной последовательности генерировался *уникальный* ключ; если это условие нарушается, то метод ToDictionary возбуждает исключение.

Метод ToLookup преобразует последовательность в так называемую *таблицу просмотра* — индексированную ключом типа TKey коллекцию типа ILookup<TKey, TElement>, которая отличается от словаря Dictionary тем, что с каждым ключом можно связывать *несколько* значений (т. е. *последовательность* значений), а также тем, что полученная таблица просмотра будет доступна только для чтения. Таблицы просмотра используются в реализациях методов Join и GroupJoin для повышения эффективности их работы. Приведем интерфейс таблиц просмотра:

```
public interface ILookup<TKey, TElement>
  : IEnumerable<IGrouping<TKey, TElement>>,
    IEnumerable
{
  int Count { get; }
  bool Contains(TKey key);
  IEnumerable<TElement> this[TKey key] { get; }
}
```

Для любого из вариантов методов ToDictionary и ToLookup предусмотрен перегруженный вариант с дополнительным параметром comparer типа IEqualityComparer<TKey>, позволяющим переопределить способ сравнения ключей.

Метод AsEnumerable предназначен, прежде всего, для преобразования к типу IEnumerable<T> последовательностей, используемых в других интерфейсах LINQ API (и, прежде всего, в запросах к удаленным базам данных); метод AsQueryable, наоборот, обеспечивает преобразование «обычных» последовательностей (типа IEnumerable<T>) к типу IQueryable<T>, используемому в интерфейсах LINQ API, предназначенных для работы с базами данных.

Группа 9. Поэлементные операции

```
IEnumerable<TSource> → TSource
First([TSource => bool])
FirstOrDefault([TSource => bool])
Last([TSource => bool])
LastOrDefault([TSource => bool])
Single([TSource => bool])
SingleOrDefault([TSource => bool])
ElementAt(int index)
ElementAtOrDefault(int index)
```

Запросы, содержащие эти методы, выполняются немедленно после конструирования (т. е. не являются отложенными).

Если имя метода оканчивается на OrDefault, то при отсутствии требуемого элемента метод не возбуждает исключение (в отличие от «парного» к нему метод без суффикса «OrDefault»); вместо этого он возвращает значение default(TSource).

Варианты методов First, Last, Single без лямбда-выражений возвращают первый, последний и единственный элемент входной последовательности; наличие предиката (лямбда-выражения) приводит к тому, что указанный элемент выбирается только среди элементов, удовлетворяющих предикату. Если требуемых элементов больше одного, то методы Single и SingleOrDefault возбуждают исключение.

В методах ElementAt и ElementAtOrDefault индексирование, как обычно, ведется от 0. Если входная последовательность поддерживает интерфейс IList<T>, то эти методы выполняются быстро, так как для доступа к требуемому элементу вызывается соответствующий индексатор.

Группа 10. Агрегирование

```
IEnumerable<TSource> → скалярный тип
Count([TSource => bool]) → int
LongCount([TSource => bool]) → long
Average([TSource => числовой_тип])→double или decimal
Sum([TSource => числовой_тип]) → числовой_тип
Max() → TSource
Max(TSource => TResult) → TResult
Min() → TSource
Min(TSource => TResult) → TResult
Aggregate((TSource, TSource) => TSource) → TSource
Aggregate(TResult seed,
  (TResult, TSource) => TResult) → TResult
Aggregate(TAccumulate seed, (TAccumulate, TSource)
  => TAccumulate, TAccumulate => TResult) → TResult
```

Методы Count и LongCount возвращают количество элементов во входной коллекции; при наличии дополнительного параметра-предиката возвращается количество элементов, удовлетворяющих указанному предикату. Если входная последовательность реализует интерфейс ICollection<T>, то для подсчета числа элементов вызывается метод Count данного интерфейса; в противном случае выполняется перебор элементов.

Методы Average и Sum могут вызываться без параметра, если элементы входной последовательности имеют числовой тип int, long, float, double, decimal. Метод Sum возвращает значение, тип которого совпадает с типом элементов последовательности или, при наличии лямбда-выражения, — с типом, возвращаемым лямбда-выражением. Метод Average возвращает значение double при обработке числовых данных любых типов, кроме decimal; для данных типа decimal возвращаемый результат также имеет тип decimal. Наряду с любым из указанных числовых типов в методах Sum и Average можно использовать его Nullable-варианты (int?, long?, float?, double?, decimal?); в этом случае возвращаемое значение тоже будет иметь Nullable-тип (при этом в случае вычисления среднего значения элементы, равные null, не учитываются).

Методы Max и Min могут вызываться без параметра, если тип элементов входной последовательности реализует интерфейс IComparable<T>. Этот же интерфейс должен реализовываться типом TResult, используемым в лямбда-выражении.

Метод Aggregate предназначен для реализации *нестандартного агрегирования*. В любом его варианте в качестве одного из параметров указывается лямбда-выражение с двумя параметрами (seed, elem) определяющее, каким образом к уже имеющемуся значению *аккумулятора* seed будет добавляться

значение очередного элемента elem исходной последовательности типа IEnumerable<TSource>. Если это лямбда-выражение является единственным параметром, то лямбда-выражение обрабатывает все элементы последовательности, начиная со второго, а первый элемент последовательности используется в качестве параметра seed при обработке второго элемента. В данном случае все параметры и возвращаемое значение лямбда-выражения должны иметь тип TSource. Пример — вычисление факториала (метод Range описан в группе 12):

```
static int Fact1(int n)
{
  return Enumerable.Range(1, n) // или Range(2, n-1)
    .Aggregate((seed, elem) => seed * elem);
}
```

Эта реализация вычисляет правильные значения факториала, только если они не превосходят int.MaxValue.

Первый вариант метода Aggregate может быть также использован, например, для нахождения минимального или максимального элемента последовательности, если сравнение элементов должно проводиться нестандартным образом. Пример — нахождение первой строки максимальной длины:

```
static string MaxLength(Enumerable<string> s)
{
  return s.Aggregate((seed, elem) =>
    seed.Length > elem.Length ? seed : elem);
}
```

Попытка использования выражения s.Max() приведет к неверному результату, поскольку в этом случае будет выполняться *лексикографическое* сравнении строк.

Во втором варианте метода Aggregate начальное значение аккумулятора задается в качестве первого параметра, а лямбда-выражение обрабатывает *все* элементы последовательности, включая и первый. Аккумулятор может иметь тип, отличный от типа TSource, причем тип возвращаемого значения будет совпадать с типом аккумулятора. Пример — более надежный вариант вычисления факториала:

```
static double Fact2(int n)
{
  return Enumerable.Range(1, n)
    .Aggregate(1.0, (seed, elem) => seed * elem);
}
```

Например, Fact1(32) вернет неправильное значение –2147483648, а Fact2(32) — хотя и приближенное, но правильное по порядку величины значение 2.63130836933694E+35. что Fact2(100) вернет значение double.PositiveInfinity.

В третьем, наиболее гибком варианте метода Aggregate накопление результата выполняется так же, как и во втором, однако предусмотрено еще одно лямбда-выражение, предназначенное для преобразования полученного значения аккумулятора к окончательному результату. При этом результат может иметь тип, отличающийся и от типа элементов обрабатываемой последовательности, и от типа аккумулятора. Пример — реализация метода расширения (аналогичного методу Join класса string), выполняющего объединение строковых представлений всех элементов последовательности, при котором между соседними элементами помещается указанная строка-разделитель separator (которая может быть пустой).

```
public static ExtCombine
{
  public static string Combine<T>(this
    IEnumerable<T> src, string separator)
  {
    return src.Aggregate("", (seed, s) => seed +
      s.ToString() + separator,
        s => separator.Length > 0 ?
          s.Remove(s.Length - separator.Length) : s);
  }
}
```

Группа 11. Квантификаторы

```
IEnumerable<TSource> → bool
All(TSource => bool)
Any([TSource => bool])
Contains(TSource value)
SequenceEqual(IEnumerable<TSource> second)
```

Метод All возвращает true, если *все* элементы последовательности удовлетворяют указанному параметру-предикату; метод Any возвращает true, если *какие-либо* элементы последовательности удовлетворяют указанному параметру-предикату (при отсутствии параметра метод Any возвращает true, если последовательность является непустой).

Метод Contains возвращает true, если в последовательности имеется *хотя бы один* элемент со значением value; метод SequenceEqual возвращает true, если вызвывшая его последовательность совпадает с последовательностью second (последовательности считаются равными, если они содержат одни и те же элементы в том же самом порядке). Для методов Contains и SequenceEqual предусмотрен перегруженный вариант с дополнительным параметром comparer типа IEqualityComparer<TSource>, позволяющим переопределить способ сравнения элементов последовательности на равенство.

Группа 12. Генерирование последовательностей

```
Enumerable → последовательность
Empty<TResult>() → IEnumerable<TResult>
Repeat<TResult>(TResult element, int count)
  → IEnumerable<TResult>
Range(int start, int count) → IEnumerable<int>
```

Метод Empty создает пустую последовательность типа IEnumerable<TResult>, метод Repeat создает последовательность типа IEnumerable<TResult>, содержащую count копий элемента element; метод Range создает числовую последовательность, содержащую count последовательных целых чисел, начиная с числа start.

Эти методы не являются методами расширения, поэтому их надо вызывать как статические методы класса Enumerable.

Метод Empty оказывается полезным в ситуации, когда какие-либо из обрабатываемых коллекций могут оказаться отсутствующими, т. е. равными null (не следует путать эту ситуацию с ситуацией, когда коллекция существует, но является пустой). Пример:

```
string[] src = { "ABC", null, "10" };
var res = src.SelectMany(s => s);
foreach (var e in res)
  Console.WriteLine(e);
// будет возбуждено исключение
```

Исправление — операция ?? совместно с методом Empty:

```
var res = src.SelectMany(s => s ??
  Enumerable.Empty<char>());
foreach (var e in res)
  Console.WriteLine(e);
// A, B, C, 1, 0
```

При обработке null-строки возвращается пустая коллекция, которая никак не влияет на выходную последовательность, но предотвращает возбуждение исключения.