Лекция №5. Элементы функционального программирования и LINQ

Функциональное программирование — парадигма программирования, в которой программы создаются последовательным применением функций. На основе функционального подхода разработаны различные языки программирования, например F#, который так же как и C# работает на платформе .NET. Так же элементы функционального программирования добавляют и не в функциональные языки, поэтому будет полезно познакомится с некоторыми концепциями.

Чистые функции

Программа в функциональном стиле — программа, состоящая из **чистых функций**.

Чистыми называют функции, которые являются детерминированными и не имеют побочных эффектов.

Детерминированность. Для одного и того же набора входных значений функция возвращает одинаковый результат.

Отсутствие побочных эффектов. Функция вообще никак не взаимодействует с внешним миром, кроме вызова других чистых функций и возврата своего результата: не может менять переданные ей аргументы, не может менять глобальные переменные, не может писать или читать что-то с консоли.

```
Func<int, int> f = x => 3 * x; // чистая

// остальные примеры не чистые
Func<int, int> f1 = x => x + i; // зависит не только от своих аргументов
Func<int, int> f2 = x => x + new Random().Next(); // не детерминирована
Func<int, int> f3 = x => { Console.Write("!"); return x; }; // влияет на
внешний мир
Func<int[], int> f4 = x => { x[0] = 3; return 3; }; // меняет переданный
аргумент
```

Некоторые следствия чистоты

1. Если результат чистой функции не используется, ее вызов может быть удален без вреда для других выражений.
2. Замена вызова чистой функции на ее результат никак не поменяет работу программы. Вместо вызова `f(3)` из примера можно сразу написать константу `9`.
3. Если функции не зависят друг от друга по данным, их можно выполнять в любом порядке. Например, в выражении `f(a) + g(b)` можно вычислить сначала `f`,

потом `g`, можно наоборот, а можно вообще одновременно (на двух ядрах).

Достоинства чистых функций

1. Повышение надежности кода за счет четкой структуры и отсутствия необходимости отслеживания побочных эффектов.
2. Удобство тестирования за счет отсутствия побочных эффектов.
3. Возможность оптимизации при компиляции, т.к. последовательность вызовов функций не фиксирована.
4. Широкие возможности для автоматического распараллеливания вычислений.

Недостатки чистых функций

1. Лишнее выделение памяти из-за неизменяемости аргументов и отсутствия глобальных переменных. Высокая нагрузка на сборщик мусора.
2. Функции ввода-вывода, работы с сетью и подобные не могут быть чистыми.

Советы по применению чистых функций

В списке достоинств чистоты надежность, понятность кода и удобство тестирования. В некоторых случаях это можно получить без усилий. Несколько советов:

- Избегайте побочных эффектов. Предположим, метод помимо основного действия производит еще и побочный эффект. Значит, он делает 2 действия. Может, получится его разбить на 2 метода, один из которых будет чистый. Либо получится передать в метод выше лишнюю ответственность, например, запись на консоль.
- Не меняйте переданные в метод входные аргументы. Например, будет неожиданным, если после выполнения `GetMedian(numbers)` массив чисел окажется отсортирован.
- Используйте неизменяемые структуры данных. Тот же `GetMedian` может вместо `List` принимать `IReadonlyList`, тогда компилятор не даст изменить элементы списка. Еще пример: Класс `Vector` можно реализовать так, чтобы все операции изменения создавали новый объект `vector`, а не меняли текущий. Это похоже на то, что происходит в строках.
- Метафору чистоты можно перенести на классы. Вызовы методов чистого класса должны менять только его состояние, но не окружающий мир. Результат методов должен зависеть только от переданных аргументов и внутреннего состояния объекта. Такой класс будет независим от остальной части программы и его легко будет тестировать.
- Передавайте в linq-методы только чистые функции. Другие программисты не ожидают побочных эффектов в этом месте. Для грязных дел есть for. Методы linq ленивые, поэтому их цепочка выполняется задом наперед. Вообще не разобраться, если при этом еще будут побочные эффекты.

LINQ

`LINQ` (**Language-Integrated Query**) — это встроенный в C# механизм для удобной работы с коллекциями, который реализует многие концепции функционального программирования.

Большинство алгоритмов, которые на менее развитых языках принято писать с помощью циклов и условных операторов, более компактно и красиво выражаются с помощью примитивов `LINQ`.

Посмотрите на код поиска всех новых писем в классическом стиле:

```
public List<int> GetNewLetterIds_ClassicWay()
{
    var res = new List<int>();
    for(int i=0; i<letters.Length; i++)
    {
        if (letters[i].IsNew)
            res.Add(letters[i].Id);
    }
    return res;
}
```

Похожий код каждому программисту приходилось писать не один раз.
А вот версия решения той же задачи с помощью `LINQ`:

```
public IEnumerable<int> GetNewLetterIds_LinqWay()
{
    return letters.Where(letter => letter.IsNew).Select(letter =>
letter.Id);
}
```

Всего одна строчка! Короткая и после некоторого привыкания понятная.

В основе `LINQ` лежит интерфейс **последовательности** `IEnumerable<T>`.
Последовательность — это абстракция чего-то, что можно начать перечислять и переходить от текущего элемента к следующему пока последовательность не закончится (или пока не надоест).

Массивы, `List`, `Dictionary`, `HashSet` — все эти коллекции реализуют интерфейс последовательности.

Для `IEnumerable<T>` в пространстве имен `System.Linq` определено множество полезных методов расширения, которые и образуют основу `LINQ`.

Фильтрация и преобразование

`Where` используется для фильтрации перечисляемого. Он принимает в качестве параметра функцию-предикат и возвращает новое перечисляемое, состоящее только из тех элементов исходного перечисляемого, на которых предикат вернул `true`.

Вот его полная сигнатура:

```
IEnumerable<T> Where(this IEnumerable<T> items, Func<T, bool> predicate)
```

`Select` используется для поэлементного преобразования перечисляемого. Он принимает в качестве параметра преобразующую функцию и возвращает новое перечисляемое, полученное применением этой функции к каждому элементу исходного перечисляемого.

```
IEnumerable<R> Select(this IEnumerable<T> items, Func<T, R> map)
```

Самое время еще раз взглянуть на предыдущий пример — его логика должна проясниться:

```
public IEnumerable<int> GetNewLetterIds()
{
    return letters
        .Where(letter => letter.IsNew) // Оставили только новые письма
        .Select(letter => letter.Id);  // Каждое оставшееся письмо
превратили в его идентификатор
}
```

Take, Skip, ToArray, ToList

Пора познакомиться еще с несколькими простыми, но часто используемыми методами.

`Take` обрезает последовательность после указанного количества элементов.

```
IEnumerable<T> Take(this IEnumerable<T> items, int count)
```

`Skip` обрезает последовательность, пропуская указанное количество элементов с начала.

```
IEnumerable<T> Skip(this IEnumerable<T> items, int count)
```

`ToArray` и `ToList` используются для преобразования `IEnumerable<T>` в массив `T[]` или в `List<T>`, соответственно.

Эти методы, как и предыдущие, не меняют исходную коллекцию, а возвращают новую последовательность.

Пример, показывающий работу всех этих методов вместе:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

IEnumerable<int> even = numbers.Where(x => x % 2 == 0);
IEnumerable<int> squares = even.Select(x => x * x);
IEnumerable<int> squaresWithoutOne = squares.Skip(1);
IEnumerable<int> secondAndThirdSquares = squaresWithoutOne.Take(2);
int[] result = secondAndThirdSquares.ToArray();

// `Assert.That` — это метод библиотеки NUnit. Он проверяет истинность
некоторого условия.
// В данном случае, что result — это массив из двух элементов 16 и 36.
Assert.That(result, Is.EqualTo(new[] { 16, 36 }));
```

Method chaining

Несколько последовательных действий с перечисляемым можно объединять в одну цепочку вызовов. Такой прием называется Method Chaining. Однако для улучшения читаемости вашего кода настоятельно рекомендуется каждый вызов метода помещать в отдельную строку, вот так:

```
Assert.That(
    new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
        .Where(x => x % 2 == 0)
        .Select(x => x * x)
        .Skip(1)
        .Take(2)
        .ToArray(),
    Is.EqualTo(new[] { 16, 36 }));
```

Method chaining делает код компактнее, но скрывает информацию о типах и семантике промежуточных значений. Иногда всё же стоит оставлять вспомогательные переменные, чтобы сделать код более читаемым.

SelectMany

Этот метод несколько менее очевиден, чем предыдущие, однако он довольно часто пригождается в самых разных задачах.

```
IEnumerable<R> SelectMany(this IEnumerable<T> items, Func<T, IEnumerable<R>> f)
```

В качестве аргумента он принимает функцию, преобразующую каждый элемент исходной последовательности в новую последовательность. А результатом работы является конкатенация всех полученных последовательностей.

Следующий пример пояснит работу этого метода:

```
string[] words = {"ab", "", "c", "de"};
IEnumerable<char> letters = words.SelectMany(w => w.ToCharArray());
Assert.That(letters, Is.EqualTo(new[] {'a', 'b', 'c', 'd', 'e'}));
```

Впрочем строка уже сама по себе является последовательностью символов и реализует интерфейс `IEnumerable<char>`, поэтому вызов `ToCharArray` на самом деле лишний.

```
string[] words = {"ab", "", "c", "de"};
var letters = words.SelectMany(w => w); // <= исчез вызов ToCharArray
Assert.That(letters, Is.EqualTo(new[] {'a', 'b', 'c', 'd', 'e'}));
```

OrderBy и Distinct

Для сортировки последовательности в `LINQ` имеется четыре метода:

```
IOrderedEnumerable<T> OrderBy<T>(this IEnumerable<T> items, Func<T, K>
keySelector)
IOrderedEnumerable<T> OrderByDescending<T>(this IEnumerable<T> items,
Func<T, K> keySelector)
IOrderedEnumerable<T> ThenBy<T>(this IOrderedEnumerable<T> items, Func<T, K>
keySelector)
IOrderedEnumerable<T> ThenByDescending<T>(this IOrderedEnumerable<T> items,
Func<T, K> keySelector)
```

Первые два дают на выходе последовательность, упорядоченную по возрастанию/убыванию ключей. А `keySelector` — это как раз функция, которая каждому элементу последовательности ставит в соответствие некоторый ключ, по которому его будут сравнивать при сортировке.

```
var names = new[] { "Pavel", "Alexander", "Anna" };

IOrderedEnumerable<string> sorted = names.OrderBy(n => n.Length);

Assert.That(sorted, Is.EqualTo(new[] { "Anna", "Pavel", "Alexander" }));
```

Если при равенстве ключей вы хотите отсортировать элементы по другому критерию, на помощь приходит метод ThenBy.

Например, в следующем примере все имена сортируются по убыванию длин, а при равных длинах — лексикографически.

```
var names = new[] { "Pavel", "Alexander", "Irina" };
var sorted = names
    .OrderByDescending(name => name.Length)
```

```
    .ThenBy(n => n);
Assert.That(sorted, Is.EqualTo(new[] { "Alexander", "Irina", "Pavel"
}).AsCollection);
```

Чтобы убрать из последовательности все повторяющиеся элементы, можно воспользоваться функцией `Distinct`.

```
var numbers = new[] { 1, 2, 3, 3, 1, 1, };
var uniqueNumbers = numbers.Distinct();
Assert.That(uniqueNumbers.Count(), Is.EqualTo(3));
```

Функции агрегирования

В `LINQ` есть удобные методы для вычисления минимума, максимума, среднего и количества элементов в последовательности.

Вот все они в действии:

```
IEnumerable<int> nums = new int[] {8, 9, 0, 1, 2, 3, 4, 5, 6, 7};

Assert.That(nums.Count(), Is.EqualTo(10));

Assert.That(nums.Min(), Is.EqualTo(0));

string[] words = { "hi", "kitty" };

Assert.That(words.Select(word => word.Length).Max(), Is.EqualTo(5));
// Можно записать строку выше короче, если воспользоваться другой
перегрузкой функции агрегирования.
// Подобные перегрузки есть у всех функций агрегирования.
Assert.That(words.Max(word => word.Length), Is.EqualTo(5));

Assert.That(nums.Average(n => n*n), Is.EqualTo(28.5));
```

Все эти методы при вызове полностью обходят коллекцию. Исключение составляет только метод `Count` — если последовательность на самом деле реализует интерфейс `ICollection` (в котором есть свойство `Count`), то `LINQ`-метод `Count()` не станет перебирать всю коллекцию, а сразу вернет значение свойства `Count`.

Благодаря этой оптимизации, временная сложность работы `LINQ`-метода `Count()` на массивах, списках, хеш-таблицах и многих других структурах данных — `O(1)`.

Есть еще две полезные функции: `All` и `Any`, которые проверяют, выполняется ли заданный предикат для всех элементов последовательности или хотя бы для одного элемента соответственно.

```
int[] numbers = {1, 2, 6, 2, 8, 0, 10, 6, 1, 2};

Assert.That(numbers.All(n => n >= 0), Is.EqualTo(true));
Assert.That(numbers.All(n => n%2 == 0), Is.EqualTo(false));

Assert.That(numbers.Any(n => n == 0), Is.EqualTo(true));
Assert.That(numbers.Any(n => n < 0), Is.EqualTo(false));
```

Группировка

`LINQ` содержит несколько методов группировки элементов последовательности по некоторому признаку. Основной способ группировки — это метод `GroupBy`. Вот его полная сигнатура:

`IEnumerable<IGrouping<TKey, TItem>> GroupBy(this IEnumerable<TItem> items, Func<TItem, TKey> keySelector)`

`keySelector` по каждому элементу последовательности получает значение ключа. Все элементы последовательности с одинаковым значением ключа образуют группу.

Пример ниже показывает, как можно разбить список имен в группы по первой букве имени:

```
string[] names = {"Pavel", "Peter", "Andrew", "Anna", "Alice", "John"};
IGrouping<char, string>[] groups = names
    .GroupBy(name => name[0])
    .OrderBy(group => group.Key)
    .ToArray();
// Каждая группа IGrouping реализует интерфейс IEnumerable:
string[] firstGroup = groups[0].OrderBy(name => name).ToArray();
Assert.That(firstGroup, Is.EqualTo(new[] {"Alice", "Andrew",
"Anna"}).AsCollection);

// Кроме того у группы есть поле Key, в котором хранится общий для этой
группы ключ группировки
char firstKey = groups[0].Key;
Assert.That(firstKey, Is.EqualTo('A'));
```

В некотором смысле `GroupBy` — это метод противоположный по действию методу `SelectMany`. `GroupBy` создает группы, а `SelectMany` из списка групп делает плоский список.

`SelectMany` после `GroupBy` не поменяют состав последовательности, но могут изменить порядок следования элементов:

```
string[] names = {"Pavel", "Peter", "Andrew", "Anna", "Alice", "John"};
var names2 = names
    .GroupBy(name => name[0])
    .SelectMany(group => group);

// Is.Equivalent игнорирует порядок элементов при сравнении коллекций:
Assert.That(names2, Is.EquivalentTo(names));
```

ToDictionary и ToLookup

Нередко встречается необходимость, сгруппировав элементы, преобразовать их в структуру данных для поиска группы по ключу группировки.

Это можно было бы сделать с помощью такой комбинации:

```
string[] names = {"Pavel", "Peter", "Andrew", "Anna", "Alice", "John"};

var namesByLetter = new Dictionary<char, List<string>>();
foreach (var group in names.GroupBy(name => name[0]))
    namesByLetter.Add(group.Key, group.ToList());

Assert.That(namesByLetter['J'], Is.EquivalentTo(new[] { "John" }));
Assert.That(namesByLetter['P'], Is.EquivalentTo(new[] {"Pavel", "Peter"}));
Assert.IsFalse(namesByLetter.ContainsKey('Z'));
```

Ровно того же эффекта можно добиться и без цикла при помощи `LINQ`-метода `ToDictionary`:

`IDictionary<K, V> ToDictionary(this IEnumerable<T> items, Func<T, K> keySelector, Func<T, V> valueSelector)`

```
string[] names = {"Pavel", "Peter", "Andrew", "Anna", "Alice", "John"};

Dictionary<char, List<string>> namesByLetter = names
    .GroupBy(name => name[0])
    .ToDictionary(group => group.Key, group => group.ToList());

Assert.That(namesByLetter['J'], Is.EquivalentTo(new[] { "John" }));
Assert.That(namesByLetter['P'], Is.EquivalentTo(new[] {"Pavel", "Peter"}));
Assert.IsFalse(namesByLetter.ContainsKey('Z'));
```

Но еще проще воспользоваться специальным методом `ToLookup`:

- `ILookup<K, T> ToLookup(this IEnumerable<T> items, Func<T, K> keySelector)`
- `ILookup<K, V> ToLookup(this IEnumerable<T> items, Func<T, K> keySelector, Func<T, V> valueSelector)`

```
string[] names = {"Pavel", "Peter", "Andrew", "Anna", "Alice", "John"};
ILookup<char, string> namesByLetter = names.ToLookup(name => name[0], name
=> name.ToLower());

Assert.That(namesByLetter['J'], Is.EquivalentTo(new[] {"john"}));
Assert.That(namesByLetter['P'], Is.EquivalentTo(new[] {"pavel", "peter"}));

// Lookup по неизвестному ключу возвращает пустую коллекцию.
//Часто это удобнее, чем поведение Dictionary, который в такой ситуации
бросает исключение.
Assert.That(namesByLetter['Z'], Is.Empty);
```