

Советы по использованию LINQ

[LINQ \(../base/level1/info_linq.php\)](#) --- [LINQ to Objects \(../level1/linq_index.php\)](#) --- [Советы по использованию LINQ](#)

Несмотря на то что доступно много очень полезных ресурсов для разработчиков, которые желают изучить весь потенциал LINQ, ниже я приведу несколько небольших советов, которые помогут начать.

Используйте ключевое слово `var`, когда запутались

Ключевое слово `var` необходимо использовать при захвате последовательности от анонимных классов в переменную, иногда это удобный способ заставить код компилироваться, когда возникает путаница со сложными обобщенными типами. Хотя предпочтительнее подход к разработке, при котором точно известно, какого типа данные содержатся в последовательности — в том смысле, что для `IEnumerable<T>` должен быть известен тип `T` — иногда, особенно в начале работы с LINQ, это может вводить в заблуждение. Если обнаруживается, что код не компилируется из-за несоответствия типов данных, попробуйте заменить явно установленные типы переменных на указанные с применением ключевым словом `var`.

Например, предположим, что есть следующий код:

C#

```
// Этот код не компилируется.
Northwind db =
    new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

IEnumerable<> orders = db.Customers
    .Where(c => c.Country == "USA" && c.Region == "WA")
    .SelectMany(c => c.Orders);
```

Может быть неясно, каков тип данных у последовательности `IEnumerable`. Вы знаете, что это `IEnumerable` некоторого типа `T`, но что собой представляет `T`? Удобный трюк состоит в присваивании результата запроса переменной, тип которой указан с помощью ключевого слова `var`, и затем получить тип текущего значения переменной, так что тип `T` известен:

C# тест (легкий) (<https://professorweb.ru/test/c-sharp-test.html>)

.NET тест (средний) (<https://professorweb.ru/test/asp-test.html>)

C#

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");

var orders = db.Customers
    .Where(c => c.Country == "USA" && c.Region == "WA")
    .SelectMany(c => c.Orders);

Console.WriteLine(orders.GetType());
```

Используйте операции Cast или OfType для унаследованных коллекций

Вы обнаружите, что большинство стандартных операций запросов LINQ могут быть вызваны на коллекциях, реализующих интерфейс `IEnumerable<T>`. Ни одна из унаследованных коллекций C# из пространства имен `System.Collections` не реализует `IEnumerable<T>`. Поэтому возникает вопрос: как использовать LINQ с унаследованными коллекциями?

Есть две стандартные операции запросов, специально предназначенные для этой цели - **Cast** и **OfType**. Обе они могут использоваться для преобразования унаследованных коллекций в последовательности `IEnumerable<T>`. Ниже показан пример:

C#

```
// Унаследованная коллекция
ArrayList arr = new ArrayList();
arr.Add("one");
arr.Add("two");
arr.Add("three");

// Приведем коллекцию к типу IEnumerable с помощью LINQ
IEnumerable<string> numbers1 = arr.Cast<string>().Where(n => n.Length < 4);

// То же самое с помощью операции OfType
IEnumerable<string> numbers2 = arr.OfType<string>().Where(n => n.Length < 4);
```

Разница между двумя операциями состоит в том, что `Cast` пытается привести все элементы в коллекции к указанному типу, помещая их в выходную последовательность. Если в коллекции есть объект типа, который не может быть приведен к указанному типу, генерируется исключение. Операция `OfType` пытается поместить в выходную последовательность только те элементы, которые могут быть приведены к указанному типу. Отсюда вытекает следующее правило:

C# test (легкий) (<https://professorweb.ru/test/csharp-test.html>)

NET test (средний) (<https://professorweb.ru/test/asp-test.html>)

Отдавайте предпочтение операции OfType перед Cast

Одной из наиболее важных причин добавления обобщений в C# была необходимость предоставить языку возможность создавать коллекции со статическим контролем типов. До появления обобщений приходилось создавать собственные специфические типы коллекций для каждого типа данных, которые нужно было в них хранить — отсутствовал способ гарантировать, что каждый элемент, помещаемый в унаследованную коллекцию, имеет один и тот же корректный тип. Ничто не могло помешать коду добавить объект TextBox в ArrayList, предназначенный для хранения только объектов Label.

С появлением обобщений в версии C# 2.0 разработчики получили в свои руки способ явно устанавливать, что коллекция может содержать только элементы заданного типа. Хотя операции OfType и Cast могут работать с унаследованными коллекциями, Cast требует, чтобы каждый объект в коллекции относился к правильному типу, что было фундаментальным недостатком унаследованных коллекций, из-за которого появились обобщения.

Когда используется операция Cast и любой из объектов в коллекции не может быть приведен к указанному типу данных, генерируется исключение. С другой стороны, с помощью операции OfType в выходной последовательности IEnumerable<T> будут сохранены только объекты указанного типа, и никаких исключений генерироваться не будет. При лучшем сценарии все объекты относятся к правильному типу, поэтому все попадают в выходную последовательность. В худшем сценарии некоторые элементы будут пропущены, но в случае применения операции Cast они привели бы к исключению.

Не рассчитывайте на безошибочность запросов

Запросы LINQ часто являются отложенными и не выполняются сразу при вызове. Например, рассмотрим следующий фрагмент кода:

C#

```
string[] greetings = {"one" , "two", "Hello LINQ :)"};
```

```
var items =
```

```
    from s in greetings
```

```
    where s.EndsWith("LINQ :)")
```

```
    select s;
```

```
foreach (var item in items)
```

```
    Console.WriteLine(item) ;
```

C# тест (легкий) (<https://professorweb.ru/test/c-sharp-test.html>)

.NET тест (средний) (<https://professorweb.ru/test/asp-test.html>)

Хотя может показаться, что запрос выполняется при инициализации переменной `items`, на самом деле это не так. Поскольку операции `Where` и `Select` являются отложенными, запрос на самом деле не выполняется в этой точке. Запрос просто вызывается, объявляется или определяется, но не выполняется. Все начинает происходить тогда, когда из него извлекается первый результат. Это обычно происходит при перечислении переменной с результатами запроса. В этом примере результат запроса не востребован до тех пор, пока не запустится оператор `foreach`. Такое поведение запроса позволяет называть его **отложенным**.

Очень легко забыть о том, что многие операции запросов являются отложенными и не выполняются до тех пор, пока не начнется перечисление результатов. Это значит, что можно иметь неправильно написанный запрос, который сгенерирует исключение только тогда, когда начнется перечисление его результатов. Такое перечисление может начаться намного позже, так что можно легко забыть, что причиной неприятностей стал неправильный запрос.

Рассмотрим код:

C#

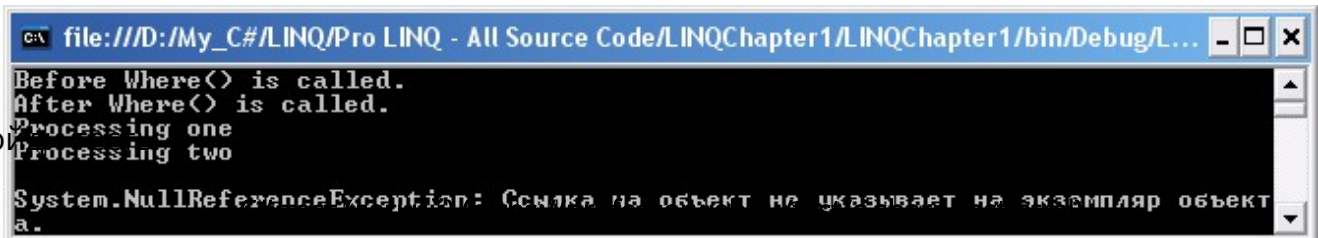
```
string[] strings = { "one", "two", null, "three" };

Console.WriteLine("Before Where() is called.");
IEnumerable<string> ieStrings = strings.Where(s => s.Length == 3);
Console.WriteLine("After Where() is called.");

foreach (string s in ieStrings)
{
    Console.WriteLine("Processing " + s);
}
```

Известно, что третий элемент в массиве строк — `null`, и нельзя вызвать `null.Length` без генерации исключения. Выполнение кода благополучно пройдет строку, где вызывается запрос. Все будет хорошо до тех пор, пока не начнется перечисление последовательности `ieStrings`, и не дойдет до третьего элемента, где возникнет исключение. Ниже показан результат выполнения этого кода:

Прой



```
C:\ file:///D:/My_C#/LINQ/Pro LINQ - All Source Code/LINQChapter1/LINQChapter1/bin/Debug/L...
Before Where() is called.
After Where() is called.
Processing one
Processing two
System.NullReferenceException: Ссылка на объект не указывает на экземпляр объект
a.
```

.NET тест (средний) (<https://professorweb.ru/test/asp-test.html>)

Как видите, вызов операции `Where` прошел без исключения. Оно не появилось до тех пор, пока при перечислении не была произведена попытка обратиться к третьему элементу последовательности. Теперь представьте, что последовательность `ieStrings` передана функции, которая дальше выполняет перечисление последовательности — возможно, чтобы наполнить раскрывающийся список или какой-то другой элемент управления. Легко подумать, что исключение вызвано сбоем в этой функции, а не самим запросом LINQ.

Используйте преимущество отложенных запросов

Следует отметить, что отложенный запрос, который в конечном итоге возвращает `IEnumerable<T>`, может перечисляться снова и снова, получая последние данные из источника. В этом случае не нужно ни вызывать, ни, как отмечалось ранее, объявлять запрос заново.

В большинстве примеров кода вы увидите вызов запроса и возврат `IEnumerable<T>` для некоторого типа `T`, сохраняемый в переменной. Затем обычно запускается оператор `foreach` на последовательности `IEnumerable<T>`. Это реализовано для демонстрационных целей. Если код выполняется много раз, повторный вызов запроса — лишняя работа. Более оправданным может быть наличие метода инициализации запроса, который вызывается однажды в жизненном цикле контекста, и в котором конструируются все запросы. Затем можно выполнить перечисление конкретной последовательности, чтобы получить последнюю версию результатов из запроса.

Используйте свойство `Log` из `DataContext`

При работе с LINQ to SQL не забывайте, что класс базы данных, генерируемый `SQLMetal`, унаследован от `System.Data.Linq.DataContext`. Это значит, что сгенерированный класс `DataContext` имеет некоторую полезную встроенную функциональность, такую как **свойство `Log`** типа `TextWriter`.

Одна из полезных возможностей объекта `Log` состоит в том, что он выводит эквивалентный SQL-оператор запроса `IQueryable<T>` до подстановки параметров. Случалось ли вам сталкиваться с отказом кода в рабочей среде, который, как вам кажется, вызван данными? Не правда ли, было бы хорошо запустить запрос на базе данных, вводя его в `SQL Enterprise Manager` или `Query Analyzer`, чтобы увидеть в точности, какие данные он возвращает? Свойство `Log` класса `DataContext` выводит

Пройди тесты

C# тест (легкий) (<https://professorweb.ru/test/c-sharp-test.html>)

C# .NET тест (средний) (<https://professorweb.ru/test/asp-test.html>)

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");

db.Log = Console.Out;

IEnumerable<Order> orders = from c in db.Customers
                           from o in c.Orders
                           where c.Country == "USA" && c.Region == "WA"
                           select o;

foreach (Order item in orders)
    Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

Назад (1_1.php)	1	2	3	Вперед (1_3.php)
-----------------	---	---	---	------------------



Лучший чат для C# программистов (<https://t.me/professorweb>)

Professor Web (/)

Наш любимый хостинг (/)

Пройди тесты

C# тест (легкий) (<https://professorweb.ru/test/c-sharp-test.html>)

.NET тест (средний) (<https://professorweb.ru/test/asp-test.html>)