ГЛАВА 15

Лямбда-выражения

Большинство новых средств C# 3.0 открывают программистам на C# мир выразительного функционального программирования. Функциональное программирование в его чистом виде — это методология программирования, построенная на основе неизменяемых переменных (иногда называемых *символами*), функций, которые могут производить другие функции, и рекурсии; и это лишь несколько его основ. К выдающимся языкам функционального программирования можно отнести Lisp, Haskell, F#[1] и Scheme[2]. Однако функциональное программирование не требует применения чистого функционального языка, и его с успехом можно реализовать на традиционных императивных языках, таких как все C-подобные языки (включая C#). Средства, добавленные в C# 3.0, трансформируют язык в более выразительный гибридный язык, в котором приемы императивного и функционального программирования сосуществуют в гармонии друг с другом. Лямбда-выражения — несомненно, самый большой кусок этого пирога функционального программирования.

Введение в лямбда-выражения

Лямбда-выражения позволяют кратко определять функциональные объекты для использования в любой момент. В языке C# эта возможность всегда поддерживалась через делегаты, с помощью которых можно создать функциональный объект (в форме делегата) и привязать к нему код обратного вызова во время создания. Лямбда-выражения связывают эти два действия — создание и подключение — в один выразительный оператор кода. Вдобавок с функциональными объектами легко ассоциировать окружение, используя конструкцию под названием *замыкание* (closure). *Функционал* (functional) — это функция, принимающая функции в своем списке параметров и оперирующая этими функциями, возможно, даже возвращая другую функцию в результате. Например, функционал может принимать две функции, одна из которых выполняет одну математическую операцию, а другая — другую математическую операцию, и возвращать третью функцию, представляющую собой комбинацию первых двух. Лямбда-выражения предлагают более естественный способ создания и вызова функционалов.

---

[1] F# — замечательный новый язык функционального программирования для .NET Framework. Подробную информацию о нем можно получить в книге Роберта Пикеринга (Robert Pickering) *Foundations of F#* (Apress, 2007 г.).

[2] Те, кто знаком с метапрограммированием на C++, определенно знакомы и с приемами функционального программирования. Если вы используете C++ и интересуетесь метапрограммированием, почитайте блестящую книгу Дэвида Абрахамса (David Abrahams) и Алексея Гуртового (Aleksey Gurtovoy) *C++ Template Metaprogramming* (Addison-Wesley Professional, 2004 г.).

В простых синтаксических терминах лямбда-выражение — это синтаксис, посредством которого можно объявлять анонимные функции (делегаты) более гладким и выразительным образом. В то же время, как будет показано далее, их возможности намного более многогранны. Практически каждое использование анонимного метода может быть заменено лямбда-выражением. Вообще говоря, нет причин, по которым нельзя внедрить технику функционального программирования в C# 2.0[3]. Однако синтаксис лямбда-выражений может потребовать некоторого времени на привыкание. С другой стороны, синтаксис лямбда-выражений очень прямолинеен. Тем не менее, встроенные в код лямбда-выражения иногда не совсем просто расшифровать.

Лямбда-выражения принимают две формы. Форма, которая является наиболее прямой заменой анонимных методов по синтексису, представляет собой блок кода, заключенный в фигурные скобки. Эту форму лямбда-выражения иногда называют *лямбда-оператором*. Она является прямой заменой анонимных методов. Лямбда-выражения, с другой стороны, предлагают еще более краткий способ объявления анонимного метода и не требуют ни кода в фигурных скобках, ни оператора возврата. Оба типа лямбда-выражений могут быть преобразованы в делегаты. Однако лямбда-выражения без блоков операторов представляют собой нечто действительно впечатляющее. С помощью типов из пространства имен `System.Linq.Expressions` их можно преобразовать в деревья выражений. Другими словами, функция, описанная в коде, превращается в данные. Тема создания деревьев выражений из лямбда-выражений раскрывается в разделе "Деревья выражений" далее в главе.

Лямбда-выражения и замыкания

Для начала рассмотрим простейшую форму лямбда-выражений, не содержащую блока операторов. Как упоминалось в предыдущем разделе, лямбда-выражение — это сокращенный способ объявления простого анонимного метода. Следующее лямбда-выражение может быть использовано в качестве делегата, принимающего один параметр и возвращающего результат деления значения параметра на 2:

```
x => x / 2
```

Это выражение говорит следующее: "взять `x` в качестве параметра и вернуть результат следующей операции на `x`". Обратите внимание, что лямбда-выражение лишено информации о типе. Это не значит, что выражение не имеет типа. Вместо этого компилятор выводит тип аргумента и тип результата из контекста его использования. Отсюда следует, что если лямбда-выражение присваивается делегату, то типы определения делегата используются для определения типов внутри лямбда-выражения. Ниже показан код, в котором лямбда-выражение присваивается типу делегата:

```
using System;
using System.Linq;

public class LambdaTest
{
    static void Main() {
        Func<int, double> expr = x => x / 2;
        int someNumber = 9;
        Console.WriteLine( "Результат: {0}", expr(someNumber) );
    }
}
```

[3] Некоторые примеры функционального программирования с использованием анонимных методов приведены в главе 14.

Лямбда-выражение выделено полужирным. Func<> — это вспомогательный тип, предоставленный в пространстве имен System, который можно использовать для объявления простых делегатов, принимающих до четырех аргументов и возвращающих результат. В данном случае объявляется переменная expr, которая является делегатом, принимающим int и возвращающим double. Когда компилятор присваивает лямбда-выражение переменной expr, он использует информацию о типе делегата для определения того, что типом x должен быть int, а типом возврата — double.

Если запустить этот код на выполнение, можно заметить, что результат не совсем точен — было произведено округление. Этого следовало ожидать, поскольку результат x/2 представлен как int, который затем приводится к double. Для исправления ситуации необходимо соответствующим образом указать типы в объявлении делегата:

```
using System;
using System.Linq;
public class LambdaTest
{
    static void Main() {
        Func<double, double> expr = (double x) => x / 2;
        int someNumber = 9;
        Console.WriteLine( "Результат: {0}", expr(someNumber) );
    }
}
```

Для целей демонстрации в лямбда-выражение включено то, что называется *явно типизированным списком параметров*, и в этом случае x имеет тип double. Также обратите внимание, что типом expr теперь является Func<double, double>, а не Func<int, double>. Компилятор требует, чтобы при использовании типизированного списка параметров в лямбда-выражении и присваивании его делегату типы аргументов делегата в точности совпадали с этим списком. Однако поскольку int явно преобразуем в double, допускается передать someNumber в expr во время вызова, как было показано.

На заметку! Следует отметить, что типизированный список параметров должен быть заключен в скобки. Скобки также требуются при объявлении делегата, принимающего или больше одного параметра, или вообще не принимающего параметров. На самом деле скобки можно использовать всегда; они не обязательны в лямбда-выражениях с только одним типизированным параметром.

Когда лямбда-выражение присваивается делегату, тип возврата выражения обычно выводится из типов аргументов. Поэтому в следующем фрагменте кода типом возврата выражения будет double, поскольку предполагаемый тип параметра x — double:

```
Func<double, int> expr = (x) => x / 2; // Ошибка компиляции!!!
```

Но так как тип double не может быть неявно преобразован в int, компилятор выдает ошибку:

```
error CS1662: Cannot convert 'lambda expression' to
    delegate type 'System.Func<double,int>' because some of the return types
    in the block are not implicitly convertible to the delegate return type
```

ошибка CS1662: Не удается преобразовать лямбда-выражение в тип делегата System.Func<double,int>, т.к. некоторые из типов возврата в блоке не являются неявно преобразуемыми к возвращаемому типу делегата

Исправить это можно, добавив приведение результата лямбда-выражения к int:

```
Func<double, int> expr = (x) => (int) x / 2;
```

На заметку! Явные типы в списке параметров лямбда-выражения необходимы, если делегат, которому оно присваивается, имеет параметры `out` или `ref`. Кто-то может сказать, что явная фиксация типов параметров внутри лямбда-выражений лишает его элегантности и выразительной мощи. И это определенно затрудняет чтение кода.

Теперь рассмотрим простое лямбда-выражение, не принимающее параметров:

```
using System;
using System.Linq;
public class LambdaTest
{
    static void Main() {
        int counter = 0;
        WriteStream( () => counter++ );
        Console.WriteLine( "Финальное значение счетчика: {0}", counter );
    }
    static void WriteStream( Func<int> generator ) {
        for( int i = 0; i < 10; ++i ) {
            Console.Write( "{0}, ", generator() );
        }
        Console.WriteLine();
    }
}
```

Обратите внимание, насколько просто с помощью лямбда-выражения передать функцию в качестве параметра в метод `WriteStream`. Более того, переданная функция захватывает окружение, внутри которого она выполняется, а именно — значение `counter` в `Main`. Эта захваченное окружение и функция вместе обычно называется *замыканием* (closure).

И, наконец, ниже приведен пример лямбда-выражения, принимающего более одного параметра:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class LambdaTest
{
    static void Main() {
        var teamMembers = new List<string> {
            "Lou Loomis",
            "Smoke Porterhouse",
            "Danny Noonan",
            "Ty Webb"
        };
        FindByFirstName( teamMembers, "Danny",
                         (x, y) => x.Contains(y) );
    }
    static void FindByFirstName(
                    List<string> members,
                    string firstName,
                    Func<string, string, bool> predicate ) {
        foreach( var member in members ) {
            if( predicate(member, firstName) ) {
                Console.WriteLine( member );
            }
        }
    }
}
```

В данном случае лямбда-выражение используется для создания делегата, принимающего два параметра типа `string` и возвращающего значение `bool`. Как видите, лямбда-выражение предлагает симпатичный и краткий способ создания предикатов. В разделе "Вернемся к итераторам и генераторам" далее в главе будет показана новая версия примера из главы 14, которая демонстрирует использование лямбда-выражений в качестве предикатов для создания гибких итераторов.

Замыкания в C# 1.0

В "старые добрые времена" C# 1.0 создание замыканий было болезненным процессом и приходилось поступать приблизительно так:

```
using System;
unsafe public class MyClosure
{
    public MyClosure( int* counter )
    {
        this.counter = counter;
    }
    public delegate int IncDelegate();
    public IncDelegate GetDelegate() {
        return new IncDelegate( IncrementFunction );
    }
    private int IncrementFunction() {
        return (*counter)++;
    }
    private int* counter;
}
public class LambdaTest
{
    unsafe static void Main() {
        int counter = 0;
        MyClosure closure = new MyClosure( &counter );
        WriteStream( closure.GetDelegate() );
        Console.WriteLine( "Финальное значение счетчика: {0}", counter );
    }
    static void WriteStream( MyClosure.IncDelegate incrementor ) {
        for( int i = 0; i < 10; ++i ) {
            Console.Write( "{0}, ", incrementor() );
        }
        Console.WriteLine();
    }
}
```

Посмотрите, какой объем работы потребовалось выполнить, чтобы обойтись без лямбда-выражений. Дополнительный код и прочие изменения выделены полужирным. Первая задача состоит в создании объекта для представления делегата и его окружения. В данном случае окружение — это указатель на переменную `counter` в методе `Main`. Для инкапсуляции функции и ее окружения было решено использовать класс. Обратите внимание, что для этого в классе `MyClass` применяется небезопасный код. Небезопасный код необходим при использовании указателей в C#, потому что безопасность этого кода не может быть проверена CLR[4]. Затем в методе `Main` я создал экземпляр `MyClosure` и передал делегат, созданный вызовом `GetDelegate` методу `WriteStream`.

[4] Тема написания небезопасного кода на C# выходит за рамки настоящей книги. За дополнительными деталями по этому поводу обращайтесь в документацию MSDN.

Какой объем работы! К тому же и понять такой код совсем нелегко.

На заметку! Возможно, возник вопрос, почему в предыдущем длинном примере применялся указатель, для чего компиляция должна была выполняться с включенной опцией `/unsafe`. Причина в том, что нужно было подчеркнуть, что захваченная переменная может быть изменена вне кода, использующего ее. Когда компилятор C# встречает переменную в замыкании, он делает нечто подобное, но вместо использования указателя на встреченную переменную он инициализирует общедоступное поле сгенерированного класса, реализующего замыкание, ссылкой на захваченную переменную или ее копией, если она имеет тип значения. Однако любой код, который попытается модифицировать эту переменную вне контекста замыкания, модифицирует копию внутри объекта замыкания, поскольку, в конце концов, это общедоступное поле. Приверженцы чистоты дизайна поднимут шум, поскольку общедоступные поля априори считаются злом. Однако вспомните, что это часть реализации компилятора. В действительности генерируемый компилятором класс остается "непроизносимым", в том смысле, что создавать его экземпляры в коде C# нельзя, а ввод его имени в коде приводит к синтаксической ошибке. Чтобы посмотреть, как компилятор генерирует замыкания, откройте скомпилированный код в ILDASM.

Замыкания в C# 2.0

Для уменьшения накладных расходов в C# 2.0 были введены анонимные методы. Однако они не настолько функционально выразительны, как лямбда-выражения, поскольку все еще придерживаются старого императивного стиля программирования и требуют явного указания типов в списке параметров. Вдобавок синтаксис анонимных методов довольно громоздкий. Чтобы продемонстрировать отличия в синтаксисе, в следующем коде показана реализация предыдущего примера на основе анонимных методов.

```
using System;
public class LambdaTest
{
    static void Main() {
        int counter = 0;
        WriteStream( delegate () {
                        return counter++;
                    } );
        Console.WriteLine( "Финальное значение счетчика: {0}",
                           counter );
    }

    static void WriteStream( Func<int> counter ) {
        for( int i = 0; i < 10; ++i ) {
            Console.Write( "{0}, ", counter() );
        }
        Console.WriteLine();
    }
}
```

Отличия между этим и исходным примером с лямбда-выражением выделены полужирным. Определенно, это намного яснее, чем способ, которым приходилось пользоваться во времена C# 1.0. Однако он все еще не столь выразителен и краток, как версия с лямбда-выражением. Лямбда-выражения являются элегантным средством определения потенциально очень сложных функций, которые могут быть построены даже посредством сборки вместе других функций.

На заметку! В предыдущем примере кода вы наверняка заметили последствия обращения к переменной `counter` внутри лямбда-выражения. В конце концов, `counter` — это локальная переменная внутри контекста `Main`, однако в контексте `WriteStream` на нее ссылаются при вызове делегата. В разделе "Остерегайтесь сюрпризов, связанных с захваченными переменными" главы 10 было показано, как достичь того же результата с помощью анонимных методов. В терминологии функционального программирования это называется *замыканием* (closure). По сути, всякий раз, когда лямбда-выражение содержит в себе окружение, в результате получается замыкание. Как будет показано в следующем разделе, замыкания могут быть очень полезны. Однако, применяемые неправильно, замыкания чреваты неприятными сюрпризами.

Лямбда-операторы

Все продемонстрированные до сих пор лямбда-выражения относились к типу простых выражений. Другим типом лямбда-выражений являются те, которые предпочтительнее называть лямбда-операторами. По форме они подобны лямбда-выражениями из предыдущего раздела, но с тем отличием, что построены из составного блока операторов внутри фигурных скобок. По этой причине лямбда с блоками операторов должно иметь оператор `return`. Вообще все лямбда-выражения, показанные в предыдущем разделе, могут быть преобразованы в лямбда с блоком операторов, если их просто окружить фигурными скобками и предварить оператором `return`. Например, следующее лямбда-выражение:

```
(x, y) => x * y
```

может быть переписано в лямбда с блоком операторов:

```
(x, y) => { return x * y; }
```

В таком виде лямбда с блоками операторов почти идентичны анонимным методам. Но есть одно главное отличие между лямбда с блоками операторов и простыми лямбда-выражениями. Первые могут быть преобразованы только в типы делегатов, в то время как вторые — и в делегаты, и в деревья выражений, посредством семейства типов, сосредоточенных вокруг `System.Linq.Expressions.Expression<T>`. О деревьях выражений речь пойдет в следующем разделе.

На заметку! Значительная разница между лямбда с блоками операторов и анонимными методами состоит в том, что в анонимных методах должны явно указываться типы параметров, в то время как для лямбда компилятор почти всегда может выводить типы на основе контекста. Сокращенный синтаксис, предлагаемый лямбда-выражениями, стимулирует в большей степени образ мышления и подходы функционального программирования.

Деревья выражений

Показанные до сих пор лямбда-выражения заменяли функциональность делегатов. Но это далеко не все. Дело в том, что компилятор C# также обладает способностью преобразовывать лямбда-выражения в деревья выражений на основе типов из пространства имен `System.Linq.Expressions`. В разделе "Функции как данные" далее в главе объясняется, чем они хороши. Например, ранее демонстрировалось, как преобразовать лямбда-выражение в делегат:

```
Func<int, int> func1 = n => n+1;
```

В этой строке кода выражение преобразуется в делегат, принимающий единственный параметр `int` и возвращающий `int`. Однако взгляните на следующую модификацию:

```
Expression<Func<int, int>> expr = n => n+1;
```

Вот это действительно интересно! Вместо вызываемого делегата лямбда-выражение преобразуется в структуру данных, представляющую операцию. Типом переменной `expr` является `Expression<T>`, где `T` заменяется типом делегата, в который может быть преобразовано лямбда-выражение. Компилятор замечает попытку преобразования лямбда-выражения в экземпляр `Expression<Func<int,int>>` и генерирует весь необходимый код, чтобы это произошло. В некоторый момент позже это выражение можно скомпилировать в полезный делегат, как показано в следующем примере:

```
using System;
using System.Linq;
using System.Linq.Expressions;
public class EntryPoint
{
   static void Main() {
      Expression<Func<int, int>> expr = n => n+1;
      Func<int, int> func = expr.Compile();
      for( int i = 0; i < 10; ++i ) {
         Console.WriteLine( func(i) );
      }
   }
}
```

Выделенная полужирным строка отражает шаг, на котором выражение компилируется в делегат. Совсем несложно представить, что перед компиляцией дерево выражения можно было бы модифицировать, или даже скомбинировать несколько деревьев выражений для создания более сложных деревьев выражений. Также можно определить новый язык выражений или реализовать анализатор для существующего языка выражений. Фактически, когда лямбда-выражение присваивается экземпляру типа `Expression<T>`, компилятор работает как анализатор выражений. "За кулисами" он генерирует код для построения дерева выражений, который можно просмотреть с помощью ILDASM или Reflector. Предыдущий пример может быть переписан без использования лямбда-выражений, как показано ниже:

```
using System;
using System.Linq;
using System.Linq.Expressions;
public class EntryPoint
{
   static void Main() {
      var n = Expression.Parameter( typeof(int), "n" );
      var expr = Expression<Func<int,int>>.Lambda<Func<int,int>>(
                  Expression.Add(n, Expression.Constant(1)),
                  n );

      Func<int, int> func = expr.Compile();

      for( int i = 0; i < 10; ++i ) {
         Console.WriteLine( func(i) );
      }
   }
}
```

Выделенные полужирным строки заменяют единственную строку в предшествующем примере, где переменной `expr` присваивается лямбда-выражение `n => n+1`. Трудно не согласиться с тем, что первый пример читать гораздо легче. Однако этот длинный пример помогает выразить действительную гибкость деревьев выражений. Давайте разобьем процесс построения выражения на отдельные шаги. Сначала необходимо пред-

ставить параметры в списке параметров лямбда-выражения. В данном случае параметр всего один — переменная n. Поэтому начинаем со следующего кода:

```
var n = Expression.Parameter( typeof(int), "n" );
```

На заметку! В этих примерах для сокращения объема ввода и повышения читабельности кода используются неявно типизированные переменные. Вспомните, что переменные по-прежнему строго типизированы. Компилятор просто выводит их тип во время компиляции вместо того, чтобы требовать его явного указания.

Приведенная выше строка кода говорит о том, что нужно выражение для представления переменной по имени n, относящейся к типу int. Как известно, в простом лямбда-выражении тип может быть определен на основе предоставленного типа делегата. Теперь необходимо сконструировать экземпляр BinaryExpression, представляющий операцию сложения, как показано ниже:

```
Expression.Add(n, Expression.Constant(1))
```

Здесь утверждается, что выражение BinaryExpression должно состоять из прибавления выражения, представляющего константу, число 1, к выражению, представляющему параметр n. Суть шаблона должна быть ясна. Для создания экземпляров элементов выражения реализована некоторая форма шаблона проектирования Abstract Factory (Абстрактная фабрика). То есть создавать новый экземпляр BinaryExpression или любого другого строительного блока деревьев выражений нельзя — для этого должны использоваться статические методы класса Expression. Это обеспечивает потребителям необходимую гибкость и позволяет реализации Expression решать, какой тип в действительности нужен.

На заметку! Если посмотреть на определения BinaryExpression, UnaryExpression, ParameterExpression и т.д. в документации MSDN, то легко заметить, что у этих типов нет общедоступных конструкторов. Вместо этого для создания экземпляров типов-наследников Expression используется сам тип Expression, который реализует шаблон Abstract Factory и предоставляет статические методы для создания экземпляров типов-наследников Expression.

Теперь, когда имеется BinaryExpression, с помощью метода Expression.Lambda<> необходимо привязать выражение (в данном случае n+1) к параметрам в списке параметров (в данном случае n). Обратите внимание, что в примере используется обобщенный метод Lambda<>, так что можно создать тип Expression<Func<int,int>>. Применение обобщенной формы предоставляет компилятору больше информации о типе, чтобы перехватывать любые возможные ошибки во время компиляции, не позволяя им нарушить работу приложения во время выполнения.

Существует еще один инструмент, демонстрирующий, как выражения представляют операции в виде данных — Expression Tree Debugger Visualizer (Визуализатор отладчика деревьев выражений) в Visual Studio 2010. Если запустить предыдущий пример внутри отладчика Visual Studio Debugger, то при проходе точки, где выполняется присваивание выражения переменной expr, в окнах Autos (Автоматические переменные) и Locals (Локальные переменные) выражение будет разобрано и отображено как {n => (n + 1)}, несмотря на то, что оно имеет тип System.Linq.Expressions.Expression<System.Func<int,int>>. Это замечательное подспорье при создании сложных деревьев выражений.

На заметку! Если бы применялась необобщенная версия метода `Expression.Lambda`, то в результате получился бы экземпляр `LambdaExpression`, а не `Expression`. Тип `LambdaExpression` также реализует метод `Compile`, однако вместо строго типизированного делегата он возвращает экземпляр типа `Delegate`. Прежде можно будет обратиться к экземпляру `Delegate`, его потребуется привести к определенному типу делегата, в данном случае `Func<int, int>` или другой делегат с той же сигнатурой, либо же вызвать `DynamicInvoke` на делегате. Любой из этих способов может привести к исключению во время выполнения, если обнаружится несоответствие между выражением и типом делегата, который, по идее, оно должно генерировать.

Операции над выражениями

Теперь рассмотрим пример того, как взять дерево выражений, сгенерированное из лямбда-выражения, и модифицировать его для создания нового дерева выражений. В данном случае выражение `(n+1)` будет превращено в `2*(n+1)`:

```
using System;
using System.Linq;
using System.Linq.Expressions;
public class EntryPoint
{
   static void Main() {
      Expression<Func<int,int>> expr = n => n+1;

      // Теперь присвоить expr значение исходного
      // выражения, умноженное на 2.
      expr = Expression<Func<int,int>>.Lambda<Func<int,int>>(
             Expression.Multiply( expr.Body,
                                  Expression.Constant(2) ),
             expr.Parameters );
      Func<int, int> func = expr.Compile();
      for( int i = 0; i < 10; ++i ) {
         Console.WriteLine( func(i) );
      }
   }
}
```

Выделенные полужирным строки показывают этап, когда исходное лямбда-выражение умножается на `2`. Очень важно отметить, что параметры, переданные методу `Lambda<>` (во втором параметре), должны быть именно теми же экземплярами параметров, которые поступили из исходного выражения, т.е. `expr.Parameters`. Это — обязательное условие. Методу `Lambda<>` нельзя передавать новый экземпляр `ParameterExpression`, иначе во время выполнения возникнет исключение, подобное описанному ниже, поскольку новый экземпляр `ParameterExpression`, даже имея то же самое имя, на самом деле является совершенно другим экземпляром параметра.

```
System.InvalidOperationException: Lambda Parameter not in scope
```

```
System.InvalidOperationException: Лямбда-параметр не находится в области
определения
```

Существует много классов, унаследованных от класса `Expression`, и много статических методов создания его экземпляров и комбинации с другими выражениями. Все они здесь описываться не будут. Мельчайшие подробности о пространстве имен `System.Linq.Expressions` можно найти в документации MSDN.

Функции как данные

Если ранее приходилось иметь дело с функциональными языками вроде Lisp, то легко заметить сходство между деревьями выражений и представлением функций как структур данных в Lisp и подобных ему языках. Большинство людей сталкиваются с Lisp в академической среде, и часто изучаемые ими концепции в реальном мире оказываются неприменимыми. Но перед тем как отнести деревья выражений к одному из таких академических упражнений, имеет смысл посмотреть, насколько они могут быть полезны.

Как и можно было предположить, в контексте C# деревья выражений чрезвычайно полезны, когда применяются к LINQ. Исчерпывающее введение в язык LINQ будет предложено в главе 16, а пока стоит отметить самый важный факт: LINQ предоставляет собственный для языка и выразительный синтаксис описания операций над данными, которые невозможно естественным образом смоделировать объектно-ориентированным способом. Например, можно создать выражение LINQ для поиска в большом массиве, находящемся в памяти (или другом типе `IEnumerable`), элементов, которые соответствуют определенному шаблону. LINQ — расширяемый язык и может предоставлять средства оперирования с другими типами хранилищ, такими как XML и реляционные базы данных. В действительности C# поддерживает LINQ to SQL, LINQ to Dataset, LINQ to Entities, LINQ to XML и LINQ to Objects, которые все вместе позволяют выполнять операции LINQ на любых типах, реализующих интерфейс `IEnumerable`.

Но как деревья выражений действуют здесь? Предположим, что реализуете LINQ to SQL для запроса к реляционной базе данных. Пользовательская база может находиться на другом полушарии, и выполнение простого запроса окажется слишком дорогим. К тому трудно предположить, насколько сложным будет пользовательское выражение LINQ. Естественно, необходимо приложить все усилия, чтобы обеспечить максимальную эффективность.

Если выражение LINQ представлено в данных (как дерево выражений), а не в коде IL (как делегат), им можно оперировать. Возможно, есть алгоритм, который поможет выявить места, где должна быть проведена оптимизация, упрощающая выражение. Или, может быть, в результате анализа выражения выясняется, что все выражение может быть упаковано, отправлено по сети и полностью выполнено на стороне сервера.

Деревья выражений обеспечивают эту важную возможность. По завершении операций над данными дерево выражений можно транслировать в окончательную исполняемую операцию с помощью механизма, подобного `LambdaExpression.Compile`, и затем запустить ее. Если бы выражение изначально было доступно только в виде кода IL, то гибкость была бы существенно ограничена. Теперь вы наверняка сможете должным образом оценить действительную мощь деревьев выражений C#.

Полезные применения лямбда-выражений

После демонстрации, как выглядят лямбда-выражения, давайте рассмотрим некоторые их применения. На самом деле большинство описанных ниже примеров на C# можно реализовать с использованием анонимных методов или делегатов. Но поразительно, как простое синтаксическое дополнение к языку может открыть широчайшие возможности в плане выразительности.

Вернемся к итераторам и генераторам

В нескольких местах этой книги уже было описано создание пользовательских итераторов на C#[5]. Теперь давайте посмотрим, как для их создания использовать лямбда-выражения. Основной упор будет сделан на способе реализации алгоритма в коде, в данном случае алгоритма итерации, который затем превращается в многократно используемый метод, применяемый почти в любом сценарии.

На заметку! Те, кто программировал на C++ и знаком с применением стандартной библиотеки шаблонов (STL), увидят в этой нотации нечто знакомое. Большинство алгоритмов, определенных в пространстве имен `std` в заголовочном файле `<algorithm>`, требуют для выполнения своей работы предоставления предикатов. Когда STL впервые появилась в начале 90-х годов, она захватила сообщество программистов C++ подобно свежему бризу функционального программирования.

Далее будет показано, как выполнять итерацию по обобщенному типу, который может быть или не быть коллекцией в строгом смысле этого слова. Вдобавок можно сделать внешним поведение курсора итерации, а также доступ к текущему значению коллекции. После небольшого размышления то же самое можно сделать почти со всем из метода создания пользовательского итератора, включая тип хранящихся элементов, тип курсора, начальное состояние курсора, конечное его состояние и способ его продвижения. Все это демонстрируется в следующем примере, где производится проход по диагонали двумерного массива:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class IteratorExtensions
{
    public static IEnumerable<TItem>
        MakeCustomIterator<TCollection, TCursor, TItem>(
                    this TCollection collection,
                    TCursor cursor,
                    Func<TCollection, TCursor, TItem> getCurrent,
                    Func<TCursor, bool> isFinished,
                    Func<TCursor, TCursor> advanceCursor) {
        while( !isFinished(cursor) ) {
            yield return getCurrent( collection, cursor );
            cursor = advanceCursor( cursor );
        }
    }
}
public class IteratorExample
{
    static void Main() {
        var matrix = new List<List<double>> {
            new List<double> { 1.0, 1.1, 1.2 },
            new List<double> { 2.0, 2.1, 2.2 },
            new List<double> { 3.0, 3.1, 3.2 }
        };
        var iter = matrix.MakeCustomIterator(
                    new int[] { 0, 0 },
```

[5] В главе 9 итераторы представлены через оператор `yield`, а в разделе "Заимствование из функционального программирования" главы 14 рассматриваются пользовательские итераторы.

```
                (coll, cur) => coll[cur[0]][cur[1]],
                (cur) => cur[0] > 2 || cur[1] > 2,
                (cur) => new int[] { cur[0] + 1,
                                     cur[1] + 1 } );
        foreach( var item in iter ) {
            Console.WriteLine( item );
        }
    }
}
```

Давайте посмотрим, насколько многократно используемым является `MakeCustomIterator<>`. Общеизвестно, что на привыкание к синтаксису лямбда-выражений уходит некоторое время, и те, кто имел дело в основном с императивным стилем кодирования, могут столкнуться с определенными трудностями в его понимании. Обратите внимание, что `MakeCustomIterator<>` принимает три аргумента обобщенного типа. `TCollection` — это тип коллекции, который в рассматриваемом примере указан в точке использования как `List<List<double>>`. `TCursor` — тип курсора, в данном случае являющийся простым массивом целых чисел, который может рассматриваться как координаты переменной `matrix`. А `TItem` — тип, возвращаемый кодом через оператор `yield`. Остальные аргументы типов `MakeCustomIterator<>` — это типы делегатов, которые он использует для определения того, как выполнять итерацию по коллекции. Для начала нужен способ получения доступа к текущему элементу коллекции, который в примере выражается следующим лямбда-выражением, использующим значения внутри массива курсора для индексации элементов внутри матрицы:

```
(coll, cur) => coll[cur[0]][cur[1]]
```

Затем необходим способ определения факта достижения конца коллекции, для чего применяется следующее лямбда-выражение, которое просто проверяет, не вышел ли курсор за пределы матрицы:

```
(cur) => cur[0] > 2 || cur[1] > 2
```

И, наконец, необходимо знать, как перемещать курсор, что задается следующим лямбда-выражением, которое просто увеличивает обе координаты курсора:

```
(cur) => new int[] { cur[0] + 1, cur[1] + 1 }
```

В результате выполнения кода примера должен появиться вывод вроде приведенного ниже, который показывает, что действительно происходит перемещение по диагонали матрицы от левого верхнего угла к нижнему правому. На каждом шаге по пути `MakeCustomIterator<>` поручает работу, которую нужно выполнить, соответствующему делегату.

```
1
2.1
3.2
```

Другие реализации `MakeCustomIterator<>` могут принимать первый параметр типа `IEnumerable<T>`, который в данном примере должен быть `IEnumerable<double>`. Однако в случае установки этого ограничения все, что передается `MakeCustomIterator<>`, должно реализовывать `IEnumerable<>`. Переменная `matrix` реализует `IEnumerable<>`, но не в той форме, которую легко использовать, поскольку это `IEnumerable<List<double>>`. Вдобавок можно предположить, что коллекция реализует индексатор, как описано в разделе “Индексаторы” главы 4, но тогда это наложит ограничение на повторную применяемость `MakeCustomIterator<>` и то, какие объекты можно использовать в нем. В приведенном выше примере индексатор в действительности используется для об-

ращения к конкретному элементу, но его применение вынесено наружу и упаковано в лямбда-выражение, предназначенное для доступа к текущему элементу.

Более того, поскольку операции доступа к текущему элементу коллекции вынесены наружу, можно даже трансформировать данные в исходной переменной `matrix` в процессе итерации по ней. Например, можно было бы умножить каждое значение на `2` в лямбда-выражении, которое обращается к текущему элементу коллекции:

```
(coll, cur) => coll[cur[0]][cur[1]] * 2;
```

Только подумайте, насколько сложно было реализовать `MakeCustomIterator<>` с применением делегатов во времена C# 1.0? Именно это имеется в виду, когда говорится, что одно лишь появление синтаксиса лямбда-выражений в C# открывает разработчикам невероятные возможности.

В качестве финального примера рассмотрим случай, при котором пользовательский итератор вообще не выполняет итерации по элементам, а вместо этого использует генератор чисел, как показано ниже:

```
using System;
using System.Linq;
using System.Collections.Generic;
public class IteratorExample
{
    static IEnumerable<T> MakeGenerator<T>( T initialValue,
                                        Func<T, T> advance ) {
        T currentValue = initialValue;
        while( true ) {
            yield return currentValue;
            currentValue = advance( currentValue );
        }
    }
    static void Main() {
        var iter = MakeGenerator<double>( 1,
                                    x => x * 1.2 );
        var enumerator = iter.GetEnumerator();
        for( int i = 0; i < 10; ++i ) {
            enumerator.MoveNext();
            Console.WriteLine( enumerator.Current );
        }
    }
}
```

Запустив этот код на выполнение, можно увидеть следующий результат:

```
1
1.2
1.44
1.728
2.0736
2.48832
2.985984
3.5831808
4.29981696
5.159780352
```

Этот метод можно использовать для бесконечного выполнения, и тогда он остановится только по исключению переполнения памяти либо по принудительному завершению. Но главное то, что элементы, по которым выполняется итерация, не существуют

в коллекции; вместо этого они генерируются по мере необходимости при каждом перемещении итератора. Такую концепцию можно применять многими способами, даже реализуя генератор случайных чисел с помощью итераторов C#.

Замыкание (захват переменной) и мемоизация

В разделе "Остерегайтесь сюрпризов, связанных с захваченными переменными" главы 10 было описано, как анонимные методы могут захватывать контекст своего лексического окружения. Многие называют это *захватом переменной*. На языке функционального программирования это известно как *замыкание* (closure)[6]. Ниже показан простой пример замыкания:

```
using System;
using System.Linq;
public class Closures
{
    static void Main() {
        int delta = 1;
        Func<int, int> func = (x) => x + delta;
        int currentVal = 0;
        for( int i = 0; i < 10; ++i ) {
            currentVal = func( currentVal );
            Console.WriteLine( currentVal );
        }
    }
}
```

Переменная `delta` и делегат `func` формируют замыкание. Тело выражения ссылается на `delta` и потому должно иметь доступ к ней при последующем выполнении. Чтобы обеспечить это, компилятор "захватывает" переменную для делегата. "За кулисами" это означает, что тело делегата содержит ссылку на действительную переменную `delta`. Но обратите внимание, что `delta` относится к типу значения и находится в стеке. Компилятор должен предпринять какие-то действия, чтобы обеспечить существование переменной за пределами метода, в котором она объявлена, если есть вероятность, что делегат обратится к ней после выхода из этого контекста. Более того, поскольку захваченная переменная доступна как делегату, так и контексту, содержащему лямбда-выражение, это означает, что захваченная переменная может быть изменена вне контекста и вне связи с делегатом. По сути, доступ к переменной имеют два метода — `Main` и делегат. Такое поведение можно использовать в собственных интересах, но если его не ожидать, то оно может привести к серьезной путанице.

На заметку! В действительности при формировании замыкания компилятор C# берет все эти переменные и упаковывает в сгенерированный класс. Он также реализует делегат как метод этого класса. В очень редких случаях это придется учитывать, особенно если обнаружится влияние на производительность при профилировании.

Теперь давайте рассмотрим полезное применение замыканий. Одна из основ функционального программирования состоит в том, что сама функция трактуется как первоклассный объект, которым можно манипулировать и оперировать, а также вызывать. Ранее уже было показано, как лямбда-выражения могут быть преобразованы в деревья выражений, чтобы можно было оперировать ими, производя более или менее сложные выражения. Но один момент, о котором еще не упоминалось — это использование функ-

---

[6] Дополнительные сведения о замыканиях доступны по адресу `http://ru.wikipedia.org/wiki/Замыкание_(программирование)`.

ций в качестве строительных блоков для создания новых функций. Для быстрой иллюстрации того, что имеется в виду, рассмотрим два лямбда-выражения:

```
x => x * 3
x => x + 3.1415
```

Создадим метод, комбинирующий эти лямбда-выражения для получения составного лямбда-выражения:

```
using System;
using System.Linq;
public class Compound
{
   static Func<T, S> Chain<T, R, S>( Func<T, R> func1, Func<R, S> func2 ) {
      return x => func2( func1(x) );
   }
   static void Main() {
      Func<int, double> func = Chain( (int x) => x * 3, (int x) => x + 3.1415 );
      Console.WriteLine( func(2) );
   }
}
```

Метод `Chain<>` принимает два делегата и производит третий делегат, комбинируя первые два. В методе `Main` он используется для производства составного выражения. Делегат, который получается после вызова `Chain<>`, эквивалентен делегату, который получается при преобразовании следующего лямбда-выражения в делегат:

```
x => (x * 3) + 3.1415
```

Метод вроде этого, который способен связывать в цепочки произвольные выражения, действительно полезен, но давайте рассмотрим другие способы создания производных функций. Представим операцию, которая требует действительно длительного времени на вычисление. Примерами могут служить операции вычисления факториала или операции вычисления *n*-го числа Фибоначчи. Рассмотрим пример вычисления обратной константы Фибоначчи согласно формуле:

$$\sum_{k=1}^{\infty} \frac{1}{F_k} = 3.35988566...$$

Здесь F_k — число Фибоначчи[7].

Для вычисления этой константы сначала понадобится написать операцию вычисления *n*-го числа Фибоначчи:

```
using System;
using System.Linq;
public class Proof
{
    static void Main() {
        Func<int, int> fib = null;
        fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;
        for( int i = 30; i < 40; ++i ) {
            Console.WriteLine( fib(i) );
        }
    }
}
```

[7] См. статью Эрика Вайсштейна (Weisstein, Eric W.) “Reciprocal Fibonacci Constant” на сайте MathWorld по адресу `http://mathworld.wolfram.com/ReciprocalFibonacciConstant.html`.

Первое, что бросается в глаза при взгляде на этот код — формирование процедуры Фибоначчи, т.е. делегата `fib`. Он формирует замыкание на самом себе! Это определенно форма рекурсии с необходимым поведением. Запустив этот пример, вы заметите, насколько медленно он работает, всего лишь вычисляя числа Фибоначчи с 30-го по 39-е. В таком случае даже не стоит надеяться продемонстрировать вычисление обратной константы Фибоначчи. Медлительность кода обусловлена тем, что каждое вычисляемое число Фибоначчи требует чуть больше работы, чем вычисление двух предыдущих, и в результате затрачиваемое время растет лавинообразно.

Проблему можно решить, пожертвовав пространством в пользу времени — за счет кэширования чисел Фибоначчи в памяти. Но вместо модификации исходного выражения давайте посмотрим, как создать метод, принимающий первоначальный делегат в качестве параметра и возвращающий новый делегат, который заменяет оригинал. Конечная цель — получить возможность заменять первый делегат производным делегатом, не затрагивая код, который его использует. Один из таких приемов называется *мемоизацией* (memoization; также называемая "запоминанием")[8]. В соответствие с этим приемом, функция кэширования возвращает значения, и каждое возвращенное значение ассоциируется с входными параметрами. Прием работает только в случае, если функция не обладает энтропией — в том смысле, что для одних и тех же входных параметров она всегда возвращает один и тот же результат. Тогда перед вызовом действительной функции сначала проводится проверка, не был ли вычислен результат для данного параметра ранее, и если да, то он возвращается вместо вызова функции. Для очень сложных функций такая техника требует немного больше пространства памяти, но дает существенный выигрыш в скорости. Рассмотрим пример.

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class Memoizers
{
    public static Func<T,R> Memoize<T,R>( this Func<T,R> func ) {
        var cache = new Dictionary<T,R>();
        return (x) => {
            R result = default(R);
            if( cache.TryGetValue(x, out result) ) {
                return result;
            }
            result = func(x);
            cache[x] = result;
            return result;
        };
    }
}
public class Proof
{
    static void Main() {
        Func<int, int> fib = null;
        fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;
        fib = fib.Memoize();
        for( int i = 30; i < 40; ++i ) {
            Console.WriteLine( fib(i) );
        }
    }
}
```

[8] Мемоизация хорошо описана в блоге Веса Дайера (Wes Dyer) по адресу `http://blogs.msdn.com/wesdyer/archive/2007/01/26/function-memoization.aspx`.

Прежде всего, обратите внимание, что в `Main` добавился только один дополнительный оператор, в котором к делегату применяется метод `Memoize<>`, чтобы произвести новый делегат. Все прочее осталось без изменений, так что прозрачная взаимозаменяемость обеспечена. Метод `Memoize<>` помещает исходный делегат, который передан в аргументе `func`, в оболочку другого замыкания, включающего экземпляр `Dictionary<>` для хранения кэшированных значений данного делегата `func`. В процессе `Memoize<>` взятие одного делегата и возврат другого реализует кэш, который значительно повышает эффективность. Всякий раз, когда вызывается делегат, он сначала проверяет, нет ли в кэше ранее вычисленного значения.

Внимание! Естественно, мемоизация работает только с функциями, которые детерминировано повторяемы — в том смысле, что для одних и тех же параметров гарантируется получение одного и того же результата. Например, истинный генератор случайных чисел подвергнуть мемоизации невозможно.

Запустите предыдущий пример и увидите поразительную разницу в скорости выполнения. Теперь вполне можно приступить к вычислению обратной константы Фибоначчи, модифицировав метод `Main` следующим образом:

```
static void Main() {
   Func<ulong, ulong> fib = null;
   fib = (x) => x > 1 ? fib(x-1) + fib(x-2) : x;
   fib = fib.Memoize();
   Func<ulong, decimal> fibConstant = null;
   fibConstant = (x) => {
      if( x == 1 ) {
         return 1 / ((decimal)fib(x));
      } else {
         return 1 / ((decimal)fib(x)) + fibConstant(x-1);
      }
   };
   fibConstant = fibConstant.Memoize();
   Console.WriteLine( "\n{0}\t{1}\t{2}\t{3}\n",
                  "Номер",
                  "Фибоначчи".PadRight(24),
                  "1/Фибоначчи ".PadRight(24),
                  "Константа Фибоначчи".PadRight(24) );

   for( ulong i = 1; i <= 93; ++i ) {
      Console.WriteLine( "{0:D5}\t{1:D24}\t{2:F24}\t{3:F24}",
                     i,
                     fib(i),
                     (1/(decimal)fib(i)),
                     fibConstant(i) );
   }
}
```

Выделенный полужирным код соответствует делегату, созданному для вычисления *n*-й обратной константы Фибоначчи. Вызывая этот делегат с все большим и большим значением `x`, несложно заметить, что результат становится все ближе и ближе к обратной константе Фибоначчи. Обратите внимание на внедрение мемоизации в делегат `fibConstant`. Если этого не сделать, то по мере вызова `fibConstant` с все большими и большими значениями `x` может возникнуть переполнение стека из-за рекурсии. Легко убедиться, что мемоизация также экономит пространство стека за счет пространства кучи. В каждой строке вывода для информации выдается промежуточное значение, но самое интересное

значение находится в крайней правой колонке. Обратите внимание, что вычисление прекращается на итерации номер 93. Причина в том, что `ulong` на 94-м числе Фибоначчи переполнится. Проблему можно решить, воспользовавшись типом `BigInteger` из пространства имен `System.Numeric`. Однако это не обязательно, поскольку 93-я итерация уже дает достаточно близкое значение обратной константы Фибоначчи.

3.359885666243177553039387

Значимые разряды выделены полужирным[9]. Как видите, прием мемоизации чрезвычайно полезен. В отношении методов, принимающих функции и производящих другие функции, можно предпринять много других полезных вещей, что будет продемонстрировано в следующем разделе.

Каррирование

В предыдущем разделе, посвященном замыканиям, было показано, как создать метод, который принимает функцию, переданную в виде делегата, и производит новую функцию. Это очень мощная концепция, и мемоизация, продемонстрированная в предыдущем разделе, является примером удачного ее применения. В этом разделе рассматривается техника, называемая *каррированием* (currying)[10]. По сути, под каррированием понимается создание операции (обычно — метода), которая принимает функцию с несколькими параметрами (обычно — делегат) и производит функцию с только одним параметром.

На заметку! Программисты на C++, знакомые с библиотекой STL, несомненно, применяли операцию каррирования, когда имели дело с любой из привязок параметров, таких как `Bind1st` и `Bind2nd`.

Пусть имеется лямбда-выражение, которое выглядит следующим образом:

```
(x, y) => x + y
```

Предположим, что есть список вещественных чисел двойной точности, и требуется применить это лямбда-выражение для добавления константного значения к каждому элементу списка, в результате чего получается новый список. Было бы здорово создать новый делегат на базе исходного лямбда-выражения, в котором одна из переменных стала бы статическим значением. Это понятие называется *привязкой параметров*, и те, кто использовал STL в C++, наверняка хорошо знакомы с ним. Ниже показан пример демонстрации привязки параметров, в котором к элементам в экземпляре `List<double>` добавляется константа `3.2`:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class CurryExtensions
{
    public static Func<TArg1, TResult>
        Bind2nd<TArg1, TArg2, TResult>(
            this Func<TArg1, TArg2, TResult> func, TArg2 constant ) {
        return (x) => func( x, constant );
    }
}
```

---

[9] Дополнительные разряды обратной константы Фибоначчи можно посмотреть по адресу `http://www.research.att.com/~njas/sequences/A079586`.

[10] Более подробные сведения о каррировании доступны по адресу `http://ru.wikipedia.org/wiki/Каррирование`.

```
public class BinderExample
{
    static void Main() {
        var mylist = new List<double> { 1.0, 3.4, 5.4, 6.54 };
        var newlist = new List<double>();

        // Исходное выражение.
        Func<double, double, double> func = (x, y) => x + y;

        // Каррированная функция.
        var funcBound = func.Bind2nd( 3.2 );
        foreach( var item in mylist ) {
            Console.Write( "{0}, ", item );
            newlist.Add( funcBound(item) );
        }
        Console.WriteLine();
        foreach( var item in newlist ) {
            Console.Write( "{0}, ", item );
        }
    }
}
```

Основа этого примера — расширяющий метод `Bind2nd<>`, который выделен полужирным. Как видите, он создает замыкание и возвращает новый делегат, принимающий только один параметр. Когда этот новый делегат вызывается, он передает свой единственный параметр исходному делегату в качестве первого параметра и предоставленную константу — в качестве второго. В примере выполняется итерация по списку `myList` с построением нового списка в переменной `newList`, при этом используется каррированная версия исходного метода для добавления константы `3.2` к каждому элементу.

Для сравнения рассмотрим другой способ каррирования, несколько отличающийся от показанного в предыдущем примере:

```
using System;
using System.Linq;
using System.Collections.Generic;
public static class CurryExtensions
{
    public static Func<TArg2, Func<TArg1, TResult>>
        Bind2nd<TArg1, TArg2, TResult>(
            this Func<TArg1, TArg2, TResult> func ) {
        return (y) => (x) => func( x, y );
    }
}
public class BinderExample
{
    static void Main() {
        var mylist = new List<double> { 1.0, 3.4, 5.4, 6.54 };
        var newlist = new List<double>();

        // Исходное выражение.
        Func<double, double, double> func = (x, y) => x + y;

        // Каррированная функция.
        var funcBound = func.Bind2nd()(3.2);
        foreach( var item in mylist ) {
            Console.Write( "{0}, ", item );
            newlist.Add( funcBound(item) );
        }
        Console.WriteLine();
```

```
        foreach( var item in newlist ) {
           Console.Write( "{0}, ", item );
        }
    }
}
```

Код, отличающийся от предыдущего примера, выделен полужирным. В первом примере метод `Bind2nd<>` возвращал делегат, принимающий один параметр `int` и возвращающий значение `int`. В данном примере `Bind2nd<>` изменен так, чтобы он возвращал делегат, который принимает один параметр (значение для привязки второго параметра в исходной функции) и возвращает другой делегат, представляющий собой каррированную функцию. Обе формы совершенно допустимы. Тем не менее, приверженцы чистоты стиля могут отдать предпочтение второй форме.

Анонимная рекурсия

В разделе "Замыкание (захват переменной) и мемоизация" ранее в главе была показана форма рекурсии, использующая замыкания при вычислении чисел Фибоначчи. В продолжение дискуссии давайте рассмотрим похожее замыкание, которое можно использовать для вычисления факториала числа:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
```

Этот код работает, потому что `fact` формирует замыкание на самом себе и также вызывает себя. То есть вторая строка, в которой `fact` присваивается лямбда-выражение для вычисления факториала, захватывает сам делегат `fact`. Несмотря на то что такая рекурсия работает, она является весьма хрупкой, и во время ее применения в таком виде, как она показана, следует соблюдать предельную осторожность. Причины будут описаны ниже.

Вспомните, что невзирая на то, что замыкание захватывает переменную для использования внутри анонимного метода, который реализован здесь в виде лямбда-выражения, захваченная переменная остается доступной для изменения вне контекста захватывающего анонимного метода или лямбда-выражения. Например, посмотрим, что произойдет, если поступить следующим образом:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
Func<int, int> newRefToFact = fact;
```

Поскольку объекты в CLR являются ссылочными типами, `newRefToFact` и `fact` теперь ссылаются на один и тот же делегат. Теперь предположим, что сделано такое:

```
Func<int, int> fact = null;
fact = (x) => x > 1 ? x * fact(x-1) : 1;
Func<int, int> newRefToFact = fact;
fact = (x) => x + 1;
```

Запланированная рекурсия разрушена! Заметили, почему? Причина связана с модификацией захваченной переменной `fact`. Ей присвоена ссылка на новый делегат, основанный на лямбда-выражении `(x) => x+1`. Но `newRefToFact` по-прежнему ссылается на лямбда-выражение `(x) => x > 1 ? x * fact(x-1) : 1`. Однако когда делегат, на который ссылается `newRefToFact`, обращается к `fact`, вместо рекурсии он выполняет новое выражение `(x) => x+1`, поведение которого отличается от имевшейся ранее рекурсии. В конечном счете, проблема вызвана тем фактом, что замыкание, заключающее в себе рекурсию, позволяет модифицировать захваченную переменную (делегат `func`) извне. Если захваченная переменная изменяется, рекурсия может быть нарушена.

Существует несколько способов решить описанную проблему, но наиболее типичный состоит в использовании анонимной рекурсии[11]. Для этого лямбда-выражение вычисления факториала потребуется изменить, чтобы оно принимало еще один параметр — делегат, который должен вызываться во время рекурсии. Это приводит к удалению замыкания и превращению захваченной переменной в параметр делегата. В итоге получается нечто вроде следующего:

```
delegate TResult AnonRec<TArg,TResult>( AnonRec<TArg,TResult> f, TArg arg );
AnonRec<int, int> fact = (f, x) => x > 1 ? x * f(f, x-1) : 1;
```

Суть в том, что вместо организации рекурсии на основе захваченной переменной, которая является делегатом, делегат рекурсии передается в качестве параметра. То есть, захваченная переменная заменяется переменной, переданной в стеке (в данном случае параметром `f` делегата `fact`). В рассматриваемом примере делегат рекурсии представлен параметром `f`. Поэтому обратите внимание, что `fact` не только принимает `f` как параметр, но вызывает его для организации рекурсии и затем передает `f` в следующую итерацию делегата. По сути, захваченная переменная теперь находится в стеке, так как она передается каждой рекурсии выражения. Однако поскольку она в стеке, опасность ее модификации извне механизма рекурсии исключается.

Чтобы подробнее ознакомиться с этой техникой, почитайте статью "Anonymous Recursion in C#" ("Анонимная рекурсия в C#") в блоге Веса Дайера (Wes Dyer) по адресу `http://blogs.msdn.com/wesdyer`. В этой статье объясняется, как реализовать комбинатор неподвижной точки, обобщающий описанную выше анонимную рекурсию[12].

Резюме

В этой главе был представлен синтаксис лямбда-выражений, которые в основном являются заменой анонимных методов. Очень жаль, что лямбда-выражения не появились еще в версии C# 2.0, потому что тогда не возникла бы необходимость в анонимных методах. На примерах было показано, как преобразовать лямбда-выражения с и без тел операторов в делегаты. Вдобавок вы увидели, как лямбда-выражения без тел операторов преобразуются в деревья выражений на основе типа `Expression<T>`, определенного в пространстве имен `System.Linq.Expression`. Перед компиляцией в делегат и вызовом к деревьям выражений можно применять трансформации. В завершение главы были приведены примеры полезного применения лямбда-выражений. К ним относятся: создание обобщенных итераторов, мемоизация с использованием замыканий, привязка параметров делегатов с помощью каррирования, а также представление концепции анонимной рекурсии. Почти все перечисленные концепции являются основой функционального программирования. Несмотря на то что все эти приемы можно было реализовать в C# с использованием анонимных методов, добавление в язык лямбда-синтаксиса сделало их применение более естественным и менее сложным.

Следующая глава посвящена языку LINQ. Кроме того, будет продолжено рассмотрение связанных с ним аспектов функционального программирования.

---

[11] Дополнительные сведения об анонимной рекурсии (на английском языке) доступны по адресу `http://en.wikipedia.org/wiki/Anonymous_recursion`.

[12] Дополнительные сведения о комбинаторе неподвижной точки доступны по адресу `http://www.wikiznanie.ru/ru-wz/index.php/Комбинатор_неподвижной_точки`.