Содержание

Глава 9

Лямбда-выражения и деревья выражений

В этой главе...

- Синтаксис лямбда-выражений
- Преобразования из лямбда-выражений в делегаты
- Классы инфраструктуры для деревьев выражений
- Преобразования из лямбда-выражений в деревья выражений
- Сущность деревьев выражений
- Изменения в выведении типов и распознавании перегруженных версий

В главе 5 было показано, что версия C# 2 намного облегчает использование делегатов, благодаря неявным преобразованиям групп методов, анонимным методам и вариантности возвращаемого типа и параметров. Этого вполне достаточно для значительного упрощения и улучшения читабельности подписки на события, но делегаты в C# 2 по-прежнему остаются слишком громоздкими, чтобы ими можно было пользоваться на постоянной основе. Чтение страницы кода, переполненной анонимными методами, требует немалых усилий, к тому же вряд ли возникнет желание начать регулярно размещать несколько анонимных методов в одиночном операторе.

Одним из фундаментальных строительных блоков LINQ является возможность создания конвейеров операций наряду с любым состоянием, требуемым этими операциями. Операции могут выражать все виды логики для обработки данных: фильтрацию, упорядочение, соединение разных источников данных и многое другое. Когда запросы LINQ выполняются внутри одного процесса, такие операции обычно представляются посредством делегатов.

Операторы, содержащие несколько делегатов, характерны при манипулировании данными с помощью LINQ to Objects[1], и *лямбда-выражения* в C# 3 делают все это возможным, не принося в жертву читабельность.

Китайская грамота

Термин *лямбда-выражение* взят из *лямбда-исчисления*, которое также записывается как λ-исчисление, где λ — греческая буква, произносимая как "лямбда". Это область математики и вычислительной техники, имеющая отношение к определению и применению функций. Она существует довольно давно, и стала основой функциональных языков, таких как ML. Хорошо уже то, что для использования лямбда-выражений в C# 3 знать лямбда-исчисление не обязательно.

Выполнение делегатов — лишь часть сюжета, связанного с LINQ. Для эффективной работы с базами данных и другими механизмами запросов необходимо другое представление операций в конвейере — способ трактовки кода как данных, которые можно исследовать программно. Затем логика внутри этих операций может быть трансформирована в другую форму, такую как обращение к веб-службе, запрос SQL или LDAP — все, что подходит в конкретной ситуации.

Несмотря на возможность построения представлений для запросов в отдельном API-интерфейсе, как правило, усложняется чтение кода и утрачивается немалая часть поддержки со стороны компилятора. Здесь лямбда-выражения опять спасают положение: помимо того, что они могут применяться для создания экземпляров делегатов, компилятор C# также способен трансформировать их в *деревья выражений* (структуры данных, представляющие логику лямбда-выражений), которые могут быть проанализированы в другом коде. Словом, лямбда-выражения — это идиоматический способ представления операций в конвейерах данных LINQ, но мы будем рассматривать по одному аспекту за раз, исследуя их довольно изолированно и постепенно охватывая всю технологию LINQ.

В этой главе мы обсудим оба способа использования лямбда-выражений, хотя пока что описание деревьев выражений будет относительно элементарным, т.к. код SQL создаваться не будет. Освоив эту теорию, вы будете достаточно хорошо знать лямбда-выражения и деревья выражений к моменту, когда мы доберемся до действительно впечатляющих вопросов в главе 12.

Последний раздел этой главы посвящен исследованию изменений выведения типов в C# 3, которые в основном обусловлены появлением лямбда-выражений с неявными типами параметров. Это напоминает обучение завязыванию шнурков: сам процесс нельзя назвать захватывающим, но без такого умения вы споткнетесь, как только начнете бежать.

Давайте приступим к выяснению, как выглядят лямбда-выражения. Мы начнем с анонимного метода и постепенно трансформируем его во все более и более короткие формы.

[1] В LINQ to Objects последовательности данных обрабатываются внутри того же самого процесса. В противоположность этому, поставщики вроде LINQ to SQL выгружают такую работу во внешние по отношению к процессу системы — например, базы данных.

9.1. Лямбда-выражения как делегаты

Во многих отношениях лямбда-выражения можно рассматривать как эволюцию анонимных методов из версии C# 2. Лямбда-выражения позволяют делать почти все то, что могут делать анонимные методы, и они практически всегда более читабельны и компактны[2]. В частности, поведение захваченных переменных в лямбда-выражениях точно совпадает с их поведением в анонимных методах. В наиболее явной форме между ними нет большой разницы, но в лямбда-выражениях доступно множество сокращений, которые делают их компактными в распространенных ситуациях. Подобно анонимным методам, лямбда-выражения имеют специальные правила преобразований. Тип выражения — это не тип делегата сам по себе, но он может быть преобразован в экземпляр делегата различными путями, как неявно, так и явно. Термин *анонимная функция* охватывает анонимные методы и лямбда-выражения и во многих случаях к ним обоим применяются те же самые правила преобразований.

Мы начнем с простого примера, изначально выраженного в виде анонимного метода. Будет создан экземпляр делегата, который принимает параметр `string` и возвращает `int` (длину строки). Сначала необходимо выбрать используемый тип делегата; к счастью, в .NET 3.5 имеется целое семейство обобщенных типов делегатов, которое помогает в решении этой задачи.

9.1.1. Подготовительные работы: знакомство с типами делегатов Func<...>

В пространстве имен `System` инфраструктуры .NET 3.5 доступно пять обобщенных типов делегатов `Func`. С `Func` не связано ничего особенного — просто удобно иметь ряд предварительно определенных обобщенных типов, которые способны обрабатывать многие ситуации. Сигнатура каждого делегата принимает от нуля до четырех параметров, типы которых указаны как параметры типов[3]. Во всех случаях последний параметр типа применяется для возвращаемого типа.

Ниже перечислены сигнатуры всех типов делегатов `Func` в .NET 3.5:

```
TResult Func<TResult>()
TResult Func<T,TResult>(T arg)
TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

Например, `Func<string,double,int>` является эквивалентом типа делегата в следующей форме:

```
public delegate int SomeDelegate(string arg1, double arg2)
```

Набор делегатов `Action<...>` предоставляет эквивалентную функциональность, когда нужен возвращаемый тип `void`. Форма делегата `Action` с одним параметром существовала еще в версии .NET 2.0, но остальные появились только в .NET 3.5.

---

[2] Есть одно средство, которое доступно в анонимных методах, но не в лямбда-выражениях — возможность игнорирования параметров. Подробные сведения об этом были предоставлены в разделе 5.4.3, но на практике отсутствие данного средства для лямбда-выражений не является столь уж значимой потерей.

[3] Возможно, вы помните, что в главе 6 мы уже сталкивались с версией, вообще не принимающей параметров (но имеющей один параметр типа).

Если четырех аргументов недостаточно, версия .NET 4 дает ответ: семейства делегатов `Action<...>` и `Func<...>` в ней расширены и принимают вплоть до 16 аргументов, поэтому `Func<T1,...,T16,TResult>` имеет целых 17 параметров типов. Они главным образом призваны помочь в поддержке исполняющей среды динамического языка (Dynamic Language Runtime — DLR), с которой вы ознакомитесь в главе 14, и вам вряд ли придется иметь с ними дело напрямую.

Для рассматриваемого примера необходим тип, который принимает параметр `string` и возвращает `int`, поэтому можно воспользоваться `Func<string,int>`.

9.1.2. Первая трансформация в лямбда-выражение

Теперь, когда известен тип делегата, можно применить анонимный метод для создания экземпляра делегата. В листинге 9.1 показано создание с последующим запуском экземпляра делегата, чтобы можно было удостовериться в его работе.

Листинг 9.1. Использование анонимного метода для создания экземпляра делегата

```
Func<string,int> returnLength;
returnLength = delegate (string text) { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

Код из листинга 9.1 выводит на консоль число `5`, как и можно было ожидать. Я разделил объявление и присваивание `returnLength`, чтобы уместить код делегата в одну строку — так будет проще его отслеживать. Выражение анонимного метода выделено полужирным; это та часть, которая будет преобразована в лямбда-выражение.

Самая длинная форма лямбда-выражения выглядит так:

```
(список-явно-типизированных-параметров) => { операторы }
```

Часть `=>` представляет собой нововведение версии C# 3 и сообщает компилятору о том, что применяется лямбда-выражение. В большинстве случаев лямбда-выражения используются с типом делегата, который имеет возвращаемый тип, отличный от `void`, и когда возвращаемый результат отсутствует, синтаксис несколько менее интуитивно понятен. Это является еще одним свидетельством идиоматических изменений, произошедших в языке между версиями C# 1 и C# 3. В C# 1 делегаты обычно применялись для событий и редко что-то возвращали. В LINQ они обычно использовались как часть конвейера данных, получая входные данные и возвращая результат, который сообщает о том, что собой представляет спроецированное значение, соответствует ли элемент текущему фильтру и т.д.

Благодаря явным параметрам и операторам в фигурных скобках, эта версия выглядит очень похожей на анонимный метод. В листинге 9.2 приведен код, эквивалентный коду из листинга 9.1, но в нем применяется лямбда-выражение.

Листинг 9.2. Первое длинное лямбда-выражение, похожее на анонимный метод

```
Func<string,int> returnLength;
returnLength = (string text) => { return text.Length; };
Console.WriteLine(returnLength("Hello"));
```

И снова в коде выделено полужирным выражение, используемое для создания экземпляра делегата. При чтении лямбда-выражений удобно воспринимать часть => как "идет в", поэтому пример в листинге 9.2 можно было бы прочитать как "`text` идет в `text.Length`". Так как это единственная часть листинга, которая пока что интересует, с этого момента она будет показываться одна. Текст, выделенный в листинге 9.2 полужирным, можно заменить любыми лямбда-выражениями, перечисленными в этом разделе, и результат останется таким же.

Те же самые правила, которые управляют операторами `return` в анонимных методах, применимы и к лямбда-выражениям: нельзя возвращать значение из лямбда-выражения с возвращаемым типом `void`, а когда он не `void`, то каждый путь в коде должен возвращать значение совместимого типа[4]. Все это интуитивно понятно и редко создает препятствия.

Пока что мы не сократили код особо значительно, равно как и не улучшили каким-то образом читабельность. Давайте займемся применением сокращений.

9.1.3. Использование одиночного выражения в качестве тела

Форма, которую мы видели до сих пор, предусматривала использование полного блока кода для возврата значения. Это гибкое решение, т.к. можно иметь несколько операторов, реализовывать циклы, выполнять возврат из нескольких мест внутри блока и делать тому подобное — точно как в анонимных методах. Однако бо́льшую часть времени все тело можно легко выражать в одиночном выражении, значение которого представляет собой результат лямбда-выражения[5]. В таких случаях можно указывать только это выражение безо всяких фигурных скобок, операторов `return` или точек с запятыми. Тогда формат становится следующим:

```
(список-явно-типизированных-параметров) => выражение
```

В рассматриваемом примере это означает, что лямбда-выражение принимает такой вид:

```
(string text) => text.Length
```

Начинает выглядеть проще. А что теперь можно сказать о типе параметра? Компилятору уже известно, что экземпляры `Func<string,int>` принимают единственный параметр типа `string`, поэтому должна быть возможность указания только имени данного параметра.

9.1.4. Списки неявно типизированных параметров

Бо́льшую часть времени компилятор способен угадать типы параметров без их явного указания. В таких случаях лямбда-выражение может быть записано так:

```
(список-неявно-типизированных-параметров) => expression
```

Список неявно типизированных параметров — это просто список имен, разделенных запятыми, не содержащий типов. Указывать типы для одних параметров и не указывать для других не допускается — список в целом должен содержать либо явно типизированные, либо неявно типизированные параметры.

---

[4] Разумеется, пути в коде, генерирующие исключения, не обязаны возвращать какие-то значения, и это также касается обнаруживаемых бесконечных циклов.

[5] Этот синтаксис можно применять также и для делегата с возвращаемым типом `void`, если необходим только один оператор. По существу отбрасывается точка с запятой и фигурные скобки.

Кроме того, при наличии параметров `out` или `ref` придется использовать явную типизацию. С нашим примером в этом плане все в порядке, поэтому лямбда-выражение становится следующим:

```
(text) => text.Length
```

Теперь оно довольно короткое. Осталось не так много, от чего можно было бы избавиться. Хотя круглые скобки кажутся лишними.

9.1.5. Сокращение для единственного параметра

Когда лямбда-выражению необходим только один параметр, и он может быть неявно типизирован, C# 3 позволяет опускать круглые скобки, так что форма выглядит так:

```
имя-параметра => выражение
```

Окончательная форма рассматриваемого лямбда-выражения приобретает следующий вид:

```
text => text.Length
```

Вас может интересовать, почему с лямбда-выражениями связано настолько много частных случаев — нигде больше в языке не играет роли, например, принимает метод один параметр или больше. На самом деле то, что звучит как очень частный случай, фактически оказывается *чрезвычайно* распространенной ситуацией, а улучшения в плане читабельности, получаемые по причине устранения круглых скобок из списка параметров, могут быть значительными, когда в небольшом фрагменте кода присутствует множество лямбда-выражений.

Полезно отметить, что при желании можно поместить в круглые скобки целое лямбда-выражение — в точности как другие выражения. Иногда это способствует лучшей читабельности, скажем, когда лямбда-выражение присваивается переменной или свойству — иначе символ = может вызвать путаницу, по крайней мере, сначала. Большую часть времени все должно быть полностью читабельным вообще безо всякого дополнительного синтаксиса. В листинге 9.3 это демонстрируется в контексте первоначально кода.

Листинг 9.3. Лаконичное лямбда-выражение

```
Func<string,int> returnLength;
returnLength = text => text.Length;

Console.WriteLine(returnLength("Hello"));
```

На первых порах код в листинге 9.3 может сбивать с толку при чтении, подобно тому, как анонимные методы выглядят странными для многих разработчиков до тех пор, пока они не начнут пользоваться ими. В обычных обстоятельствах вы объявляете переменную и присваиваете ей значение в одном и том же выражении, делая его еще яснее. Однако, привыкнув к лямбда-выражениям, вы сможете оценить, насколько они лаконичны. Трудно представить себе более короткий и ясный способ создания экземпляра делегата[6].

[6] Это не значит, что сделать такое невозможно. Некоторые языки позволяют определять замыкания в виде простых блоков кода с “магическим” именем переменной для представления общего случая с единственным параметром.

Можно было бы вдобавок изменить имя переменной `text` на что-то вроде `x`, и в LINQ это часто удобно, но более длинные имена предоставляют читателю полезную информацию.

Трансформация, которая была описана на нескольких страницах, кратко проиллюстрирована на рис. 9.1; это позволяет легко оценить, насколько много излишнего синтаксиса было устранено.

Рис. 9.1. Сокращения в синтаксисе лямбда-выражений

Решение, использовать ли короткую форму для тела лямбда-выражения, указывая только выражение вместо полного блока, совершенно не зависит от решения относительно применения явных или неявных параметров. В рассмотренном примере был избран путь сокращения лямбда-выражения, но можно было бы начать с превращения параметров в неявные. Когда вы освоитесь с лямбда-выражениями, то вообще перестанете об этом думать, а будете свободно записывать кратчайшую форму из числа доступных.

Функции высшего порядка

Тело лямбда-выражения способно само содержать лямбда-выражение, и это вполне может сбивать с толку. В качестве альтернативы параметром лямбда-выражения может быть другой делегат, что в равной степени плохо. Оба примера относятся к *функциям высшего порядка*. Если вам нравится сталкиваться с запутанными ситуациями, загляните в загружаемый исходный код, сопровождающий эту книгу. Данный подход распространен в функциональном программировании и иногда оказывается удобным. Просто он требует определенной доли настойчивости в обретении правильного образа мыслей.

До сих пор мы имели дело только с одиночным лямбда-выражением, приводя его в различные формы. Давайте рассмотрим несколько примеров, чтобы конкретизировать вопросы до того, как переходить к деталям.

9.2. Простые примеры использования типа `List<T>` и событий

После того, как мы приступим к исследованию расширяющих методов в главе 10, мы будем применять лямбда-выражения постоянно. Но на данный момент наилучшие примеры можно представить посредством типа `List<T>` и обработчиков событий. Мы начнем со списков, и ради краткости будем использовать автоматически реализуемые свойства, неявно типизированные локальные переменные и инициализаторы коллекций. Затем мы будем вызывать методы, принимающие параметры в форме делегатов, конечно же, создавая делегаты с помощью лямбда-выражений.

9.2.1. Фильтрация, сортировка и действия на списках

Вспомните метод `FindAll()` в типе `List<T>` — он принимает предикат `Predicate<T>` и возвращает новый список с элементами исходного списка, соответствующими предикату. Метод `Sort()` принимает экземпляр `Comparison<T>` и сортирует список указанным образом. Наконец, метод `ForEach()` принимает действие `Action<T>` и выполняет его над каждым элементом. Для предоставления экземпляра делегата каждому из этих методов в листинге 9.4 применяются лямбда-выражения. В качестве примера данных выступают названия и годы выпуска в прокат разнообразных фильмов. Сначала на консоль выводится исходный список, затем создается и выводится отфильтрованный список, содержащий только старые фильмы, и, в конце концов, исходный список сортируется по названию фильма и снова выводится на консоль. (Небезынтересно прикинуть, насколько большой объем кода пришлось бы написать для решения этой задачи в C# 1.)

Листинг 9.4. Манипулирование списком фильмов с использованием лямбда-выражений

```
class Film
{
  public string Name { get; set; }
  public int Year { get; set; }
}
...
var films = new List<Film>
{
  new Film { Name = "Jaws", Year = 1975 },
  new Film { Name = "Singing in the Rain", Year = 1952 },
  new Film { Name = "Some like it Hot", Year = 1959 },
  new Film { Name = "The Wizard of Oz", Year = 1939 },
  new Film { Name = "It's a Wonderful Life", Year = 1946 },
  new Film { Name = "American Beauty", Year = 1999 },
  new Film { Name = "High Fidelity", Year = 2000 },
  new Film { Name = "The Usual Suspects", Year = 1995 }
};
Action<Film> print = ←❶ Создание многократно используемого делегата для вывода списка на консоль
  film => Console.WriteLine("Name={0}, Year={1}",
                            film.Name, film.Year);
films.ForEach(print);                          ←❷ Вывод на консоль исходного списка
films.FindAll(film => film.Year < 1960)        ←❸ Создание отфильтрованного списка
    .ForEach(print);
films.Sort((f1, f2) => f1.Name.CompareTo(f2.Name));   ←❹ Сортировка исходного списка
films.ForEach(print);
```

Первая часть листинга 9.4 отвечает за подготовку данных. Именованный тип применяется в коде только для простоты — анонимный тип в этом конкретном случае означал бы появление еще нескольких препятствий, которые пришлось бы преодолевать.

Перед использованием построенного списка создается экземпляр делегата ❶, который будет применяться для вывода на консоль элементов списка. Этот экземпляр делегата используется три раза, и именно потому для его хранения была создана переменная вместо того, чтобы каждый раз применять отдельное лямбда-выражение. Он просто выводит на консоль одиночный элемент, но передавая экземпляр делегата методу `List<T>.ForEach()`, можно отобразить на консоли целый список. Следует отметить один тонкий, однако важный момент — точка с запятой в конце этого оператора является частью оператора присваивания, а не лямбда-выражения. Если бы то же самое лямбда-выражение использовалось в качестве аргумента при вызове метода, то сразу после `Console.WriteLine(...)` точка с запятой бы отсутствовала.

Первым на консоль выводится исходный список безо всяких модификаций ❷. Затем в списке находятся и выводятся на консоль все фильмы, снятые до 1960 года ❸. Это делается с помощью еще одного лямбда-выражения, которое выполняется для каждого фильма в списке — оно только определяет, должен ли конкретный фильм быть включен в отфильтрованный список. В исходном коде данное лямбда-выражение указано как аргумент метода, но на самом деле компилятор создает метод, подобный показанному ниже:

```
private static bool SomeAutoGeneratedName(Film film)
{
   return film.Year < 1960;
}
```

Фактически вызов метода `FindAll()` сводится к следующему:

```
films.FindAll(new Predicate<Film>(SomeAutoGeneratedName))
```

Поддержка лямбда-выражений похожа на поддержку анонимных методов в C# 2; все основано на мастерстве компилятора. (На самом деле в этом случае компилятор Microsoft оказывается даже более интеллектуальным — он полагает, что может выиграть от повторного использования экземпляра делегата, если код обратится к нему еще раз, поэтому кеширует его.)

Сортировка списка также осуществляется с применением лямбда-выражения ❹, которое сравнивает два элемента, представляющих фильмы, по их названиям. Должен признаться, что явный вызов метода `CompareTo()` выглядит немного неуклюже. В следующей главе вы увидите, каким образом расширяющий метод `OrderBy()` позволяет выразить упорядочение в более компактной манере.

Давайте обратимся к другому примеру, на этот раз демонстрирующему использование лямбда-выражений для обработки событий.

9.2.2. Регистрация внутри обработчика событий

Возвращаясь к главе 5, в разделе 5.9 был представлен простой способ применения анонимных методов для регистрации в журнале факта возникновения событий, но компактный синтаксис можно было использовать только потому, что потеря информации, передаваемой в параметрах, тогда совершенно не беспокоила. А что, если понадобится регистрировать как природу события, так и данные о его отправителе и аргументах? Лямбда-выражения позволяют делать это компактным способом, как демонстрируется в листинге 9.5.

Листинг 9.5. Регистрация событий в журнале с применением лямбда-выражений

```
static void Log(string title, object sender, EventArgs e)
{
   Console.WriteLine("Event: {0}", title);                  // Событие
   Console.WriteLine(" Sender: {0}", sender);               // Отправитель
   Console.WriteLine(" Arguments: {0}", e.GetType());       // Аргументы
   foreach (PropertyDescriptor prop in
          TypeDescriptor.GetProperties(e))
   {
      string name = prop.DisplayName;
      object value = prop.GetValue(e);
      Console.WriteLine("    {0}={1}", name, value);
   }
}
...
Button button = new Button { Text = "Click me" };
button.Click     += (src, e) => Log("Click", src, e);
button.KeyPress  += (src, e) => Log("KeyPress", src, e);
button.MouseClick += (src, e) => Log("MouseClick", src, e);

Form form = new Form { AutoSize = true, Controls = { button } };
Application.Run(form);
```

В листинге 9.5 лямбда-выражения используются для передачи имени события *и параметров* методу `Log()`, который фиксирует в журнале детали, связанные с событием. В журнал не заносятся подробные сведения об источнике события кроме данных, возвращаемых его переопределенным методом `ToString()`, поскольку с элементами управления связано слишком много информации. Однако с помощью рефлексии по дескрипторам свойств отображаются детали переданного экземпляра `EventArgs`.

Ниже показан пример вывода, получаемого в результате щелчка на кнопке:

```
Event: Click
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
Event: MouseClick
 Sender: System.Windows.Forms.Button, Text: Click me
 Arguments: System.Windows.Forms.MouseEventArgs
   Button=Left
   Clicks=1
   X=53
   Y=17
   Delta=0
   Location={X=53,Y=17}
```

Конечно, все это *можно* было бы сделать и без лямбда-выражений, но за счет применения лямбда-выражений код получился намного более компактным.

После демонстрации преобразования лямбда-выражений в экземпляры делегатов самое время взглянуть на деревья выражений, которые представляют лямбда-выражения в виде данных, а не кода.

9.3. Деревья выражений

Идея *кода в форме данных* далеко не нова, но она не часто использовалась в популярных языках программирования. Вы могли бы привести в качестве довода то, что эта концепция применяются всеми программами .NET, т.к. код IL трактуется JIT-компилятором как данные, которые затем преобразуются в машинный код для выполнения центральным процессором. Тем не менее, это глубоко скрыто, и несмотря на существование библиотек для программного манипулирования кодом IL, используются они не особенно часто.

Деревья выражений в .NET 3.5 предлагают абстрактный способ представления определенного кода в виде дерева объектов. Он похож на модель CodeDOM, но функционирует на несколько более высоком уровне. Главной областью применения деревьев выражений является LINQ, и позже в этом разделе будет показано, насколько деревья выражений важны для LINQ в целом.

В C# 3 предоставляется встроенная поддержка преобразования лямбда-выражений в деревья выражений, но перед тем, как раскрывать ее, давайте изучим, почему они вписываются в инфраструктуру .NET Framework безо всяких трюков со стороны компилятора.

9.3.1. Построение деревьев выражений программным образом

Деревья выражений не являются чем-то мистическим, хотя некоторые случаи их использования выглядят подобными магии. Как следует из названия, они представляют собой деревья объектов, в которых каждый узел сам является выражением. Разные типы выражений представляют различные операции, которые можно выполнять в коде: бинарные операции, такие как сложение; унарные операции наподобие определения длины массива; вызовы методов; обращения к конструкторам и т.д.

Пространство имен `System.Linq.Expressions` содержит разнообразные классы, которые представляют выражения. Все они унаследованы от класса `Expression`, который является абстрактным и по большей части состоит из статических фабричных методов, предназначенных для создания экземпляров других классов выражений. Тем не менее, класс `Expression` предлагает два свойства.

- Свойство `Type` представляет тип .NET вычисляемого выражения — его можно рассматривать как возвращаемый тип. Например, типом выражения, которое извлекает свойство `Length` строки, будет `int`.
- Свойство `NodeType` возвращает вид выражения в форме члена перечисления `ExpressionType` со значениями вроде `LessThan`, `Multiply` и `Invoke`. Придерживаясь того же самого примера, в `myString.Length` часть, отвечающая за доступ к свойству, имела бы тип узла `MemberAccess`.

Существует множество классов, производных от `Expression`, и некоторые из них могут иметь множество узлов разных типов. Например, `BinaryExpression` представляет любую операцию с двумя операндами: арифметическую, логическую, сравнения, индексации массива и т.д. Именно здесь становится важным свойство `NodeType`, т.к. оно позволяет отделять различные виды выражений, которые представлены одним и тем же классом.

Я не намерен раскрывать здесь все классы выражений или типы узлов — их слишком много и они хорошо документированы в MSDN (`http://mng.bz/3vW3`). Вместо этого будет предложено общее описание того, что можно делать с деревьями выражений.

Давайте начнем с создания одного из простейших деревьев выражений, суммирующего две целочисленных константы. Код в листинге 9.6 создает дерево выражения для представления 2+3.

Листинг 9.6. Простое дерево выражения, суммирующего числа 2 и 3

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Console.WriteLine(add);
```

Запуск кода из листинга 9.6 приведет к получению вывода `(2 + 3)`, который указывает на то, что различные классы выражений переопределяют метод `ToString()` для обеспечения вывода, воспринимаемого человеком. На рис. 9.2 изображено дерево, сгенерированное кодом.

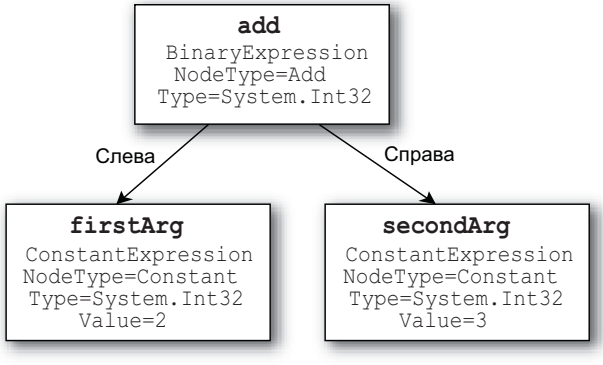

Рис. 9.2. Графическое представление дерева выражения, созданного кодом из листинга 9.6

Полезно отметить, что концевые выражения в коде создаются первыми: выражения строятся снизу вверх. Это обусловлено тем фактом, что выражения являются неизменяемыми — после того, как выражение создано, оно никогда не будет изменяться, поэтому выражения можно кешировать и многократно использовать по своему усмотрению.

Теперь, когда дерево выражения построено, наступило время его выполнить.

9.3.2. Компиляция деревьев выражений в делегаты

В число типов, производных от `Expression`, входит `LambdaExpression`. В свою очередь, от `LambdaExpression` унаследован обобщенный класс `Expression<TDelegate>`. Все это немного запутывает, поэтому с целью прояснения на рис. 9.3 показана иерархия типов.

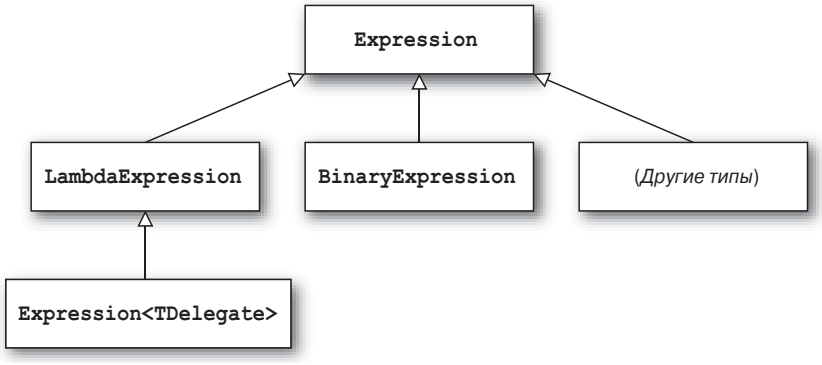

Рис. 9.3. Иерархия типов от `Expression<TDelegate>` до `Expression`

Разница между `Expression` и `Expression<TDelegate>` в том, что обобщенный класс является статически типизированным для отражения вида представляемого выражения в терминах возвращаемого типа и параметров. Очевидно, это выражается параметром типа `TDelegate`, который должен быть типом делегата. Например, простое выражение сложения не принимает параметров и возвращает целочисленное значение, что соответствует сигнатуре `Func<int>`, поэтому для представления такого выражения в статически типизированной манере можно было бы применить `Expression<Func<int>>`. Это делается с использованием метода `Expression.Lambda()`, который имеет несколько перегруженных версий. В рассматриваемых примерах применяется обобщенный метод, который использует параметр типа для указания типа представляемого делегата. С альтернативами можно ознакомиться в MSDN.

Итак, какой смысл делать все это? В классе `LambdaExpression` определен метод `Compile()`, который создает делегат подходящего типа; в `Expression<TDelegate>` имеется другой метод с таким же именем, но он статически типизирован для возвращения делегата типа `TDelegate`. Затем этот делегат может быть выполнен обычным образом, как если бы он был создан с применением нормального метода или любыми другими средствами. Сказанное иллюстрируется в листинге 9.7 на примере того же самого выражения, что и ранее.

Листинг 9.7. Компиляция и выполнение дерева выражения

```
Expression firstArg = Expression.Constant(2);
Expression secondArg = Expression.Constant(3);
Expression add = Expression.Add(firstArg, secondArg);

Func<int> compiled = Expression.Lambda<Func<int>>(add).Compile();
Console.WriteLine(compiled());
```

Код в листинге 9.7 является, вероятно, одним из самых запутанных путей вывода на консоль числа `5`, какие только можно представить. В то же время он весьма выразителен. Здесь программно создаются логические блоки и представляются в виде обычных объектов, после чего у инфраструктуры запрашивается их компиляция в действительный код, который может быть выполнен. Возможно, вам никогда не придется использовать деревья выражений подобным образом или даже вообще строить их программно, но это дает полезную справочную информацию, которая поможет лучше понять функционирование LINQ.

Как упоминалось в начале этого раздела, деревья выражений не слишком далеко ушли от модели CodeDOM — к примеру, инструмент Snippy компилирует и выполняет код C#, который вводится как простой текст. Однако между CodeDOM и деревьями выражений существуют два значительных отличия.

Во-первых, в .NET 3.5 деревья выражений обладали возможностью представления только одиночных выражений. Они не были предназначены для целых классов, методов или даже просто операторов. В .NET 4 кое-что в этом плане изменилось, и в данной версии деревья выражений применяются для поддержки динамической типизации — теперь можно создавать блоки, присваивать значения переменным и т.д. Но по сравнению с CodeDOM по-прежнему существуют значительные ограничения.

Во-вторых, в C# деревья выражений поддерживаются прямо на уровне языка через лямбда-выражения. Давайте посмотрим на это прямо сейчас.

9.3.3. Преобразование лямбда-выражений C# в деревья выражений

Как уже было указано, лямбда-выражения могут быть преобразованы в соответствующие экземпляры делегатов, либо неявно, либо явно. Это не единственное доступное преобразование. Компилятору можно также предложить построить дерево выражения из заданного лямбда-выражения, создавая экземпляр `Expression<TDelegate>` во время выполнения. Например, в листинге 9.8 показан намного более короткий путь создания выражения для "возврата числа 5", его компиляции и обращения к результирующему делегату.

Листинг 9.8. Использование лямбда-выражений для создания деревьев выражений

```
Expression<Func<int>> return5 = () => 5;
Func<int> compiled = return5.Compile();
Console.WriteLine(compiled());
```

Часть `() => 5` в первой строке листинга 9.8 — это лямбда-выражение. Никакие приведения не требуются, поскольку компилятор может проверить все самостоятельно. Вместо `5` можно было бы написать `2+3`, но компилятор применил бы к этому сложению оптимизацию, заменив его суммой. Важный момент здесь в том, что лямбда-выражение было преобразовано в дерево выражения.

Ограничения преобразований

Не *все* лямбда-выражения могут быть преобразованы в деревья выражений. Преобразовать лямбда-выражение с блоком операторов (даже если это один оператор `return`) в дерево выражения не получится — оно должно иметь форму одиночного выражения, и это выражение не может содержать присваивания. Такое ограничение применимо и в версии .NET 4 с ее расширенными возможностями в отношении деревьев выражений. Несмотря на то что это самые распространенные ограничения, существуют не только они одни — приводить здесь полный список не имеет смысла, т.к. подобного рода ситуации возникают очень редко. О наличии проблемы с преобразованием вы узнаете на этапе компиляции.

Давайте рассмотрим более сложный пример, чтобы увидеть, как все работает, особенно то, что касается параметров. На этот раз будет написан предикат, который принимает две строки и проверяет, находится ли первая строка в начале второй. Код оказывается простым, когда он представлен в виде лямбда-выражения (листинг 9.9).

Листинг 9.9. Демонстрация более сложного дерева выражения

```
Expression<Func<string, string, bool>> expression =
   (x, y) => x.StartsWith(y);

var compiled = expression.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

Это дерево выражения сложнее само по себе, особенно к тому времени, как оно преобразуется в экземпляр `LambdaExpression`. В листинге 9.10 показано, как его можно было бы построить в коде.

Листинг 9.10. Построение выражения с вызовом метода в коде

```
MethodInfo method = typeof(string).GetMethod
   ("StartsWith", new[] { typeof(string) });
var target = Expression.Parameter(typeof(string), "x");
var methodArg = Expression.Parameter(typeof(string), "y");
Expression[] methodArgs = new[] { methodArg };

Expression call =
   Expression.Call(target, method, methodArgs);

var lambdaParameters = new[] { target, methodArg };
var lambda =
    Expression.Lambda<Func<string, string, bool>>
   (call, lambdaParameters);

var compiled = lambda.Compile();

Console.WriteLine(compiled("First", "Second"));
Console.WriteLine(compiled("First", "Fir"));
```

❶ Построение частей вызова метода

❷ Создание выражения `CallExpression` из частей

❸ Преобразование в `LambdaExpression`

Как видите, объем кода в листинге 9.10 значительно превышает версию с лямбда-выражением C#. Но при этом код делает более очевидным то, что в точности происходит в дереве, и показывает, как привязываются параметры.

Сначала определяется все, что необходимо знать о вызове метода, который формирует тело финального выражения ❶: цель метода (строка, на которой вызывается `StartsWith()`); сам метод (как `MethodInfo`); и список аргументов (а этом случае содержащий всего один элемент). Так получилось, что цель и аргумент нашего метода являются параметрами, передаваемыми выражению, однако они могли быть другими типами выражений — константами, результатами других вызовов методов, значениями свойств и т.д.

После построения вызова метода как выражения ❷ нужно преобразовать его в лямбда-выражение ❸, по пути привязав параметры. В качестве информации для вызова метода повторно используются те же самые ранее созданные значения `ParameterExpression`: порядок, в котором они были указаны при создании лямбда-выражения — это порядок, в котором они будут выбираться, когда в конечном итоге вызывается делегат.

На рис. 9.4 окончательное дерево выражения представлено графически. По правде говоря, хотя это по-прежнему называется деревом выражения, факт повторного использования выражений параметров (и так должно делаться — создание нового выражения параметра с тем же самым именем и попытка привязки параметров подобным образом привела бы к генерации исключения во время выполнения) означает, что в строгом смысле оно действительно деревом не является.

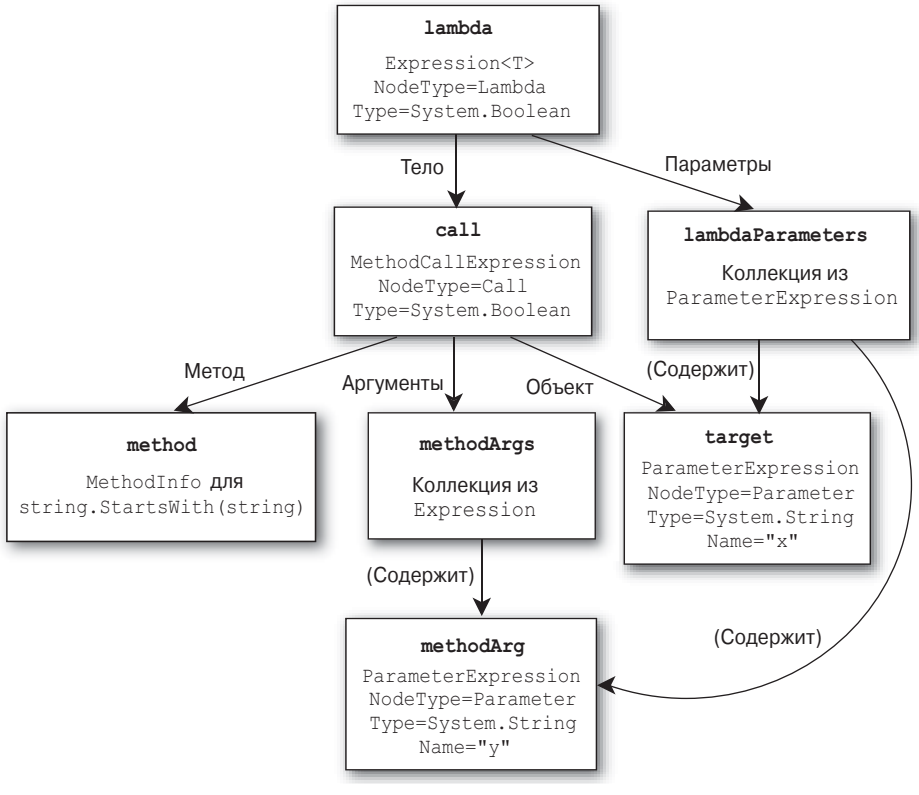

Рис. 9.4. Графическое представление дерева выражения, которое вызывает метод и использует параметры из лямбда-выражения

Бегло оценив сложность диаграммы на рис. 9.4 и кода в листинге 9.10 без попытки обратиться к деталям, можно было бы подумать, что здесь делается что-то действительно трудное, тогда как фактически это всего лишь единственный вызов метода. Представьте, как могло бы выглядеть дерево выражения для по-настоящему сложного выражения — и затем выразите благодарность за то, что версия C# 3 позволяет создавать деревья выражений из лямбда-выражений!

В качестве еще одного способа исследования той же идеи среды Visual Studio 2010 и Visual Studio 2012 предоставляют встроенный визуализатор для деревьев выражений[7]. Это может оказаться удобным, если вы пытаетесь найти способ построения дерева выражения в коде и хотите получить представление о том, как оно должно выглядеть.

[7] Если вы работаете с Visual Studio 2008, то можете загрузить из сети MSDN код примера для построения похожего визуализатора (`http://mng.bz/g6xd`), но очевидно проще воспользоваться визуализатором, входящим в состав Visual Studio, при наличии последующих версий.

Просто напишите лямбда-выражение, которое делает то, что требуется, с фиктивными данными, активизируйте визуализатор внутри отладчика и затем на основе предоставленной информации обдумайте, как построить аналогичное дерево в реальном коде. Визуализатор опирается на изменения, появившиеся в .NET 4, поэтому с проектами для целевой версии .NET 3.5 он не работает. На рис. 9.5 показано диалоговое окно визуализатора для примера с методом `StartsWith()`.

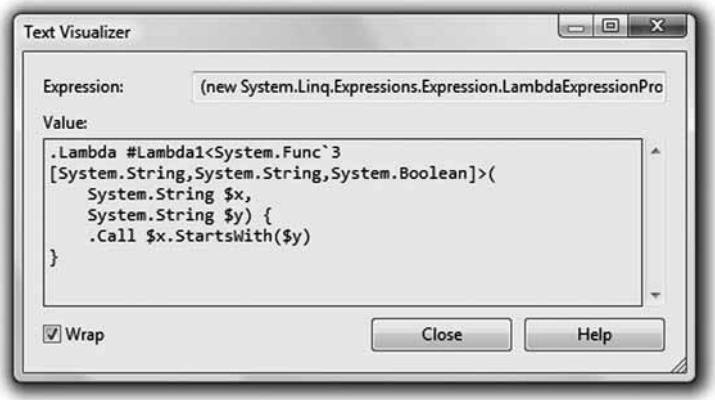

Рис. 9.5. Применение визуализатора деревьев выражений, доступного в отладчике

Части `.Lambda` и `.Call` в информации из визуализатора соответствуют вызовам методов `Expression.Lambda()` и `Expression.Call()`, а `$x` и `$y` — выражениям параметров. Результаты визуализации будут одинаковыми независимо от того, построено дерево выражения явно в коде или с использованием преобразования из лямбда-выражения.

Следует отметить один небольшой момент — хотя компилятор C# строит деревья выражений в скомпилированном коде аналогично показанному в листинге 9.10, в его арсенале имеется одно сокращение: ему не приходится применять обычную рефлексию, чтобы получить `MethodInfo` для `string.StartsWith()`. Взамен компилятор использует метод, эквивалентный операции `typeof`. Это доступно только в IL, а не в самом C#, и та же самая операция применяется для создания экземпляров делегатов из групп методов.

Разобравшись со связью между деревьями выражений и лямбда-выражениями, давайте кратко рассмотрим, по каким причинам они настолько удобны.

9.3.4. Деревья выражений являются основой LINQ

Без лямбда-выражений деревья выражений обладали бы относительно небольшой ценностью. Они могли выступать в качестве альтернативы CodeDOM в случаях, когда требовалось только моделирование одиночного выражения вместо целых операторов, методов, типов и тому подобного, но их преимущества оставались по-прежнему ограниченными.

В определенном смысле справедливо также и обратное утверждение: без деревьев выражений лямбда-выражения безусловно были бы *менее* полезными. Наличие более компактного способа создания экземпляров делегатов всегда приветствовалось, и сдвиг в сторону более функциональной формы разработки по-прежнему жизнеспособен. Как будет показано в следующей главе, лямбда-выражения особенно эффективны в сочетании с расширяющими методами, но присутствие еще и деревьев выражений делает ситуацию намного более интересной.

Что вы получите за счет комбинирования лямбда-выражений, деревьев выражений и расширяющих методов? Ответ: практически "языковую сторону LINQ". Дополнительный синтаксис, который вы увидите в главе 11, представляет собой добавочное преимущество, однако история была бы захватывающей даже только с тремя ингредиентами, упомянутыми выше. В течение долгого времени можно было иметь *либо* аккуратную проверку на этапе компиляции, *либо* возможность поручения другой платформе запуска определенного кода, обычно выражаемого в виде текста (запросы SQL являются самым очевидным примером). Тем не менее, оба средства не могли быть доступными одновременно.

Сочетая лямбда-выражения, которые предоставляют проверки на этапе компиляции, с деревьями выражений, абстрагирующими модель выполнения от заданной логики, можно в разумных пределах извлечь лучшее из обоих миров. В основе внепроцессных поставщиков LINQ лежит идея того, что дерево выражения может быть получено из знакомого исходного языка (в этом случае C#), а результат применяется как промежуточный формат, который затем преобразуется в машинный язык целевой платформы — например, SQL. В ряде ситуаций машинный язык может оказаться не настолько простым, как низкоуровневый API-интерфейс, который возможно делает разные обращения к веб-службам в зависимости от того, что именно выражение представляет. На рис. 9.6 показаны различные пути LINQ to Objects и LINQ to SQL.

В одних случаях преобразование может попробовать выполнить *всю* логику на целевой платформе, тогда как в других случаях могут использоваться возможности компиляции деревьев выражений для выполнения некоторых выражений локально и в других местах. Мы взглянем на детали этого шага преобразования в главе 12, но вы должны помнить об этой конечной цели во время исследования расширяющих методов и синтаксиса LINQ в главах 10 и 11.

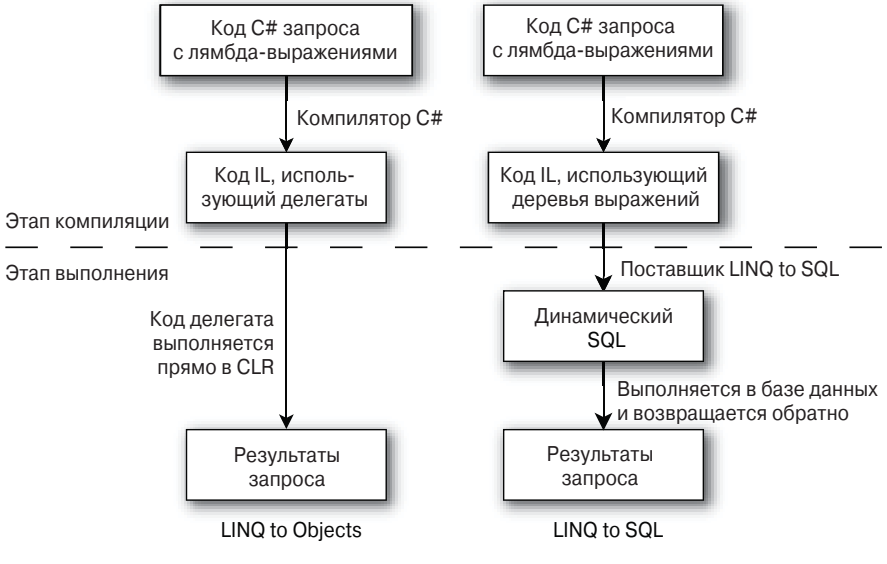

Рис. 9.6. Как LINQ to Objects, так и LINQ to SQL начинают с кода C# и в конце получают результаты запроса. Возможность выполнения кода удаленным образом появляется благодаря деревьям выражений

Не вся проверка может делаться компилятором

Когда деревья выражений анализируются преобразователем определенного вида, некоторые случаи обычно должны отбрасываться. Например, несмотря на возможность преобразования вызова метода `string.StartsWith()` в сходное SQL-выражение, вызов метода `string.IsInterned` в среде базы данных не имеет смысла. Деревья выражений обеспечивают высокую безопасность на этапе компиляции, но компилятор способен проверить только тот факт, что лямбда-выражение может быть преобразовано в допустимое дерево выражения; он не в состоянии гарантировать, что это дерево выражения окажется пригодным во всех возможных случаях его применения.

Хотя наиболее распространенные случаи использования деревьев выражений относятся к LINQ, так бывает далеко не всегда...

9.3.5. Использование деревьев выражений за рамками LINQ

Бьярне Страуструп однажды сказал: “Я бы не хотел строить инструмент, который мог бы делать только то, что я способен вообразить”. Несмотря на то что деревья выражений были введены в .NET главным образом для LINQ, с того времени и сообщество разработчиков, и проектировщики из Microsoft нашли им другие применения. Этот раздел далек от полноты, но наверняка даст вам несколько идей относительно того, в чем могут помочь деревья выражений.

Оптимизация исполняющей среды динамического языка

Исполняющая среда динамического языка (DLR) будет подробно рассматриваться в главе 14, когда речь пойдет о динамической типизации в C#, однако деревья выражений являются основной частью этой архитектуры. Деревья выражений обладают тремя характеристиками, которые делают их привлекательными для DLR.

- Они неизменяемы, поэтому их можно безопасно кешировать.
- Они объединяемы, так что можно формировать сложное поведение на основе простых строительных блоков.
- Они могут быть скомпилированы в делегаты, которые являются JIT-скомпилированными в машинный код как обычные делегаты.

Среда DLR должна принимать решения о том, как обрабатывать разнообразные выражения, в которых смысл мог быть тонко изменен, на основе различных правил. Деревья выражений позволяют этим правилам (и результатам) быть трансформированными в код, который близок к тому, что вы писали бы вручную, если бы знали все правила и результаты, показанные до сих пор. Это мощная концепция, которая обеспечивает неожиданно быстрое выполнение динамического кода.

Защищенные от рефакторинга ссылки на члены

В разделе 9.3.3 я упоминал, что компилятор может выдавать ссылки на значения `MethodInfo` почти так, как это делает операция `typeof`. К сожалению, C# не обладает такой способностью. Это означает, что единственный способ сообщить фрагменту универсального, основанного на рефлексии кода о необходимости использования свойства по имени `BirthDate`, определенного в заданном типе, ранее предусматривал применение строкового литерала и обеспечение того, что при изменении имени свойс-

тва изменялся также и этот литерал. Используя версию C# 3, можно построить дерево выражения, которое представляет ссылку на свойство с помощью лямбда-выражения. Затем метод может проанализировать дерево выражения, отыскать указанное свойство и сделать с информацией все, что необходимо. Разумеется, он может также скомпилировать дерево выражения в делегат и пользоваться им напрямую.

В качестве примера, когда такое может применяться, напишем следующий код:

```
serializationContext.AddProperty(x => x.BirthDate);
```

Это позволило бы контексту сериализации (`serializationContext`) узнать, что нужно сериализировать свойство `BirthDate`, и он мог бы записать подходящие метаданные и извлечь значение. (Сериализация — это лишь одна область, где может понадобиться ссылка на свойство или метод; такая ситуация довольно распространена внутри кода, управляемого рефлексией.) Если вы проведете рефакторинг свойства `BirthDate`, назвав его `DateOfBirth`, то лямбда-выражение также изменился. Конечно, это не защищено от неправильного использования — какая-либо проверка того, что оценка выражения действительно дает простое свойство, на этапе компиляции не предпринимается; в коде метода `AddProperty()` должна быть предусмотрена соответствующая проверка времени выполнения.

Вполне вероятно, что однажды C# позволит это делать на уровне самого языка. Операция подобного рода уже получила имя: `infoof`. На протяжении некоторого времени она находилась в списке возможных средств от команды C#, и неудивительно, что Эрик Липперт написал о ней в своем блоге (`http://mng.bz/24y7`), но эта операция пока не прошла отборочный тур. Подождем выхода версии C# 6.

Более простая рефлексия

Перед тем, как погружаться в темные глубины выведения типов, следует упомянуть еще об одном случае применения деревьев выражений, который также связан с рефлексией. Как говорилось в главе 3, арифметические операции не особенно хорошо работают с обобщениями, что затрудняет написание обобщенного кода, скажем, для сложения последовательности значений. Марк Грэвелл применяет деревья выражений для получения удивительного результата в виде обобщенного класса `Operator` и необобщенного вспомогательного класса, позволяя записывать код, подобный показанному ниже:

```
T runningTotal = initialValue;
foreach (T item in values)
{
   runningTotal = Operator.Add(runningTotal, item);
}
```

Код будет функционировать даже в случаях, когда тип значений отличается от типа накапливаемой суммы (`runningTotal`), разрешая, к примеру, добавлять целую последовательность значений `TimeSpan` к `DateTime`. Это *возможно* сделать в C# 2, но оно потребует значительно больше кропотливой работы из-за способов, по которым получается доступ к операциям через рефлексию, особенно для элементарных типов. Деревья выражений позволяют реализации этой "магии" быть довольно чистой, а тот факт, что они компилируются в обычный код IL, который затем обрабатывается JIT-компилятором, обеспечивает великолепную производительность.

Были приведены только некоторые примеры, и вне всяких сомнений множество разработчиков имеют дело с совершенно другими случаями использования деревьев выражений. Однако на этом обсуждение непосредственно лямбда-выражений и деревьев выражений закончено. Вы еще увидите их немало, когда дело дойдет до LINQ, но

прежде чем двигаться дальше, осталось рассмотреть несколько изменений языка C#, которые требуют некоторых пояснений. Эти изменения касаются выведения типов и способа выбора компилятором перегруженных версий методов.

9.4. Изменения в выведении типов и распознавании перегруженных версий

В версии C# 3 шаги, предпринимаемые в рамках выведения типов и распознавании перегруженных версий, были изменены с целью приспособления к лямбда-выражениям и обеспечения большего удобства работы с анонимными методами. Это можно не рассматривать как новое средство C# по существу, но немаловажно понимать, что компилятор собирается делать. Если вы считаете детали подобного рода скучными и несущественными, можете просто сразу переходить к чтению резюме в конце главы. Однако помните о наличии данного раздела, чтобы можно было вернуться к нему в ситуации, когда вы столкнетесь с ошибкой компиляции, связанной с этой темой, и не сумеете разобраться, почему код не заработал. (Или наоборот, может возникнуть желание прочитать его, если оказалось, что код успешно скомпилировался, в то время как вы думали, что этого не должно было произойти.)

Даже в таком разделе я не буду заглядывать в каждый укромный уголок — для этого предназначена спецификация языка; подробные сведения находятся в разделе 7.5.2 (“Type inference” (“Выведение типов”)) спецификации C# 5. Взамен я предложу обзор нового поведения, предоставляя примеры для распространенных случаев. Главная причина изменения спецификации связана с необходимостью обеспечения лаконичного стиля для записи лямбда-выражений, вот потому данная тема и включена в настоящую главу.

Давайте сначала немного подробнее ознакомимся с проблемами, которые бы возникли, если бы проектировщики из команды C# решили придерживаться старых правил.

9.4.1. Причины внесения изменений: упрощение вызова обобщенных методов

Выведение типов происходит в нескольких ситуациях. Вы уже видели его применение к неявно типизированным массивам, и оно также требуется при попытке неявного преобразования группы методов в тип делегата. Это может особенно запутывать, когда преобразование происходит при использовании группы методов в качестве аргумента другого метода. В случае перегрузки вызываемого метода *и* наличии перегруженных версий методов внутри группы методов, *а также* возможности участия обобщенных методов набор потенциальных преобразований может оказаться гигантским.

Безусловно, самая распространенная ситуация для выведения типов возникает при вызове обобщенного метода без указания любых аргументов типов. Это происходит в LINQ постоянно — способ, которым работают выражения запросов, сильно зависит от данной возможности. Все это обрабатывается настолько гладко, что очень легко проигнорировать факт выполнения компилятором большого объема работ ради придания написанному вами коду большей ясности и лаконичности.

В C# 2 правила были *умеренно* простыми, хотя группы методов и анонимные методы не всегда обрабатывались настолько хорошо, как возможно того хотелось. Процесс выведения типов не получал из них никакой информации, приводя к ситуациям, когда желаемое поведение было очевидным для разработчиков, но не для компилятора.

Из-за появления лямбда-выражений в C# 3 все еще больше усложнилось. Если вызвать обобщенный метод, используя лямбда-выражение со списком неявно типизированных параметров, компилятору придется выяснить, о каких типах идет речь, до того как он сможет проверить тело лямбда-выражения.

Суть намного проще понять, глядя на код, нежели описывая проблемы словами. В листинге 9.11 приведен пример подобного рода проблемы, на которую я ссылался: вызов обобщенного метода с применением лямбда-выражения.

Листинг 9.11. Пример кода, в котором требуются новые правила выведения типов

```
static void PrintConvertedValue<TInput,TOutput>
   (TInput input, Converter<TInput,TOutput> converter)
{
   Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

Метод `PrintConvertedValue()` в листинге 9.11 просто получает входное значение и делегат, который может преобразовать это значение в другой тип. Он полностью обобщенный — никаких предположений относительно параметров типов `TInput` и `TOutput` не делается. Теперь взгляните на типы аргументов при вызове этого метода в последней строке листинга. Первый аргумент, очевидно, имеет тип `string`, но что можно сказать о втором аргументе? Он является лямбда-выражением, поэтому его нужно преобразовать в `Converter<TInput,TOutput>`, а это означает необходимость знания типов `TInput` и `TOutput`.

Вспомните из раздела 3.3.2, что правила выведения типов в C# 2 применялись к каждому аргументу индивидуально, и не было никакого способа использования типов, выведенных для одного аргумента, во втором аргументе. В данном случае эти правила не позволили бы найти типы `TInput` и `TOutput` для второго аргумента, так что код из листинга 9.11 не смог бы компилироваться.

Наша конечная цель заключается в том, чтобы понять, что обеспечит возможность успешной компиляции кода в листинге 9.11 в C# 3, но пока начнем с чего-то более скромного.

9.4.2. Выведение возвращаемых типов анонимных функций

В листинге 9.12 представлен еще один пример кода, который по идее должен компилироваться, но это не так в условиях действия правил выведения типов C# 2.

Листинг 9.12. Попытка выведения возвращаемого типа для анонимного метода

```
delegate T MyFunc<T>();                          ← Объявление типа делегата: Func<T> отсутствует в .NET 2.0
static void WriteResult<T>(MyFunc<T> function)   ← Объявление обобщенного метода с параметром делегата
{
   Console.WriteLine(function());
}
...
WriteResult(delegate { return 5; });             ← Для T требуется выведение типа
```

При компиляции кода из листинга 9.12 с помощью компилятора C# 2 выдается сообщение об ошибке следующего вида:

```
error CS0411: The type arguments for method
'Snippet.WriteResult<T>(Snippet.MyFunc<T>)' cannot be inferred from the
usage. Try specifying the type arguments explicitly.
ошибка CS0411: Аргументы типов для метода
Snippet.WriteResult<T>(Snippet.MyFunc<T>) не могут быть выведены на основе
использования. Попробуйте указать аргументы типов явным образом.
```

Исправить ошибку можно двумя путями — либо указать аргумент типа явным образом (как предлагает компилятор), либо привести анонимный метод к конкретному типу делегата:

```
WriteResult<int>(delegate { return 5; });
WriteResult((MyFunc<int>)delegate { return 5; });
```

Оба способа работают, но выглядят неуклюжими. Желательно, чтобы компилятор выполнял такой же вид выведения типов, как в случае типов, отличных от делегатов, используя тип возвращаемого выражения для выведения типа `T`. Именно это в C# 3 делается для анонимных методов и лямбда-выражений, однако есть одна загвоздка. Хотя во многих случаях задействован только один оператор `return`, иногда их может быть больше. В листинге 9.13 показана слегка измененная версия кода из листинга 9.12, в которой анонимный метод временами возвращает целочисленное значение, а временами — объект.

Листинг 9.13. Код, возвращающий целочисленное значение или объект в зависимости от времени суток

```
delegate T MyFunc<T>();
static void WriteResult<T>(MyFunc<T> function)
{
   Console.WriteLine(function());
}
...
WriteResult(delegate
{
   if (DateTime.Now.Hour < 12)
   {
      return 10;                ← Возвращаемым типом является int
    }
   else
   {
      return new object();      ← Возвращаемым типом является object
    }
});
```

В этой ситуации для определения возвращаемого типа компилятор применяет ту же самую логику, что и в случае неявно типизированных массивов, как было описано в разделе 8.4. Он формирует набор типов из всех операторов `return`, обнаруженных в теле анонимной функции[8] (в данном случае `int` и `object`), и проверяет, есть ли среди

[8] Возвращаемые выражения, не имеющие типа, такие как `null` или другое лямбда-выражение, в этот набор не включаются. Их допустимость проверяется позже, после того как возвращаемый тип был определен, однако они не принимают участия в данном решении.

них в точности один тип, в который могут быть неявно преобразованы все остальные типы. Существует неявное преобразование из `int` в `object` (через упаковку), но не из `object` в `int`, поэтому выведенным возвращаемым типом будет `object`. Если указанному критерию не соответствует ни одного типа (или типов оказывается более одного), возвращаемый тип не может быть выведен и возникает ошибка компиляции.

Теперь вы знаете, как выяснять *возвращаемый* тип анонимной функции, но что можно сказать о лямбда-выражениях, в которых типы параметров допускают неявное определение?

9.4.3. Двухэтапное выведение типов

Детали выведения типов в C# 3 *намного* сложнее, чем в C# 2. Потребность в консультации со спецификацией относительно точного его поведения будет возникать редко, но если это все же понадобится, я рекомендую записать на бумаге все параметры типов, аргументы и прочие данные, а затем пошагово следовать спецификации, тщательно помечая каждое требуемое действие. В конечном итоге получится таблица, полная *фиксированных* и *нефиксированных* переменных типов с разными наборами *границ* для каждой из них. *Фиксированная* переменная типа — это такая переменная, для которой компилятор принял решение, касающееся ее значения; иначе она будет *нефиксированной*. *Граница* — это фрагмент информации о переменной типа. Учитывая еще и массу примечаний, я не сомневаюсь, что головная боль вам обеспечена; это не особенно интересная тема.

Я представлю менее строгий взгляд на выведение типов — он будет, скорее всего, не хуже спецификации, но намного проще для понимания. Дело в том, что если компилятор не выполняет выведение типов точно так, как вы хотите, это почти наверняка приведет к ошибке компиляции, а не к коду, который хотя и построен, но ведет себя некорректно. Если построение кода не происходит, попробуйте предоставить компилятору больше информации — это довольно просто. Ниже дано *примерное* описание того, что изменилось в версии C# 3.

Первое крупное изменение связано с тем, что в C# 3 аргументы работают слаженно, подобно единой команде. В C# 2 каждый аргумент использовался для попытки *точного* определения некоторых параметров, и компилятор предъявил бы претензии в случае, если для отдельного параметра типа любые два аргумента приводили бы к разным результатам, даже если они совместимы. В C# 3 аргументы могут нести в себе *части* информации — типы, которые должны быть неявно преобразуемыми в окончательное фиксированное значение конкретной переменной типа. Для получения этого фиксированного значения применяется та же самая логика, что и при выведении возвращаемых типов и в неявно типизированных массивах.

В листинге 9.14 показан пример, в котором не используются ни лямбда-выражения, ни даже анонимные методы.

Листинг 9.14. Гибкое выведение типов, комбинирующее информацию из множества аргументов

```
static void PrintType<T>(T first, T second)
{
  Console.WriteLine(typeof(T));
}
...
PrintType(1, new object());
```

Хотя код в листинге 9.14 *синтаксически* допустим в C# 2, он не скомпилируется; выведение типов потерпит неудачу, т.к. первый параметр предопределит, что тип `T` должен быть `int`, а второй — что `T` должен быть `object`. В C# 3 компилятор выясняет, что тип `T` должен быть `object` в точности таким же способом, как это делалось при выведении возвращаемого типа в листинге 9.13. В сущности, правила выведения возвращаемых типов являются одним примером более общего процесса в C# 3. Второе изменение заключается в том, что выведение типов теперь осуществляется в два этапа. Первый этап имеет дело с обычными аргументами, где задействованные типы известны с самого начала. Сюда входят анонимные функции со списками явно типизированных параметров.

Затем начинает действовать второй этап, на котором выводятся типы неявно типизированных лямбда-выражений и групп методов. Идея состоит в том, чтобы выяснить, достаточно ли информации, собранной компилятором из фрагментов, для определения типов параметров лямбда-выражения (или группы методов). Если ее достаточно, компилятор может приступать к исследованию тела лямбда-выражения с целью выведения возвращаемого типа, который часто является еще одним искомым параметром типа. Если второй этап предоставляет какую-то дополнительную информацию, компилятор повторяет действия второго этапа заново и так происходит до тех пор, пока не исчерпаются все возможности или не выяснятся все содержащиеся параметры типов. На рис. 9.7 сказанное представлено в виде блок-схемы, но не забывайте, что это сильно упрощенная версия алгоритма.

Давайте рассмотрим два примера, демонстрирующие работу этого алгоритма. Первым делом возьмем код, приведенный в начале этого раздела, в листинге 9.11:

```
static void PrintConvertedValue<TInput,TOutput>
   (TInput input, Converter<TInput,TOutput> converter)
{
   Console.WriteLine(converter(input));
}
...
PrintConvertedValue("I'm a string", x => x.Length);
```

Параметрами типов, которые необходимо выяснить здесь, являются `TInput` и `TOutput`. Ниже перечислены шаги, которые выполняются для этого.

1. Начинается этап 1.
2. Первый параметр имеет тип `TInput`, а первый аргумент — тип `string`. Мы делаем вывод, что должно существовать неявное преобразование из `string` в `TInput`.
3. Второй параметр имеет тип `Converter<TInput,TOutput>`, а вторым аргументом является неявно типизированное лямбда-выражение. Выведение не производится, т.к. информации для этого не достаточно.
4. Начинается этап 2.
5. Тип `TInput` не зависит от каких-либо нефиксированных параметров типов, поэтому он фиксируется как `string`.
6. Второй аргумент теперь имеет фиксированный *входной* тип, но нефиксированный *выходной* тип. Его можно рассматривать как `(string x) => x.Length` и вывести возвращаемый тип как `int`. Следовательно, неявное преобразование должно происходить из `int` в `TOutput`.
7. Этап 2 повторяется.
8. Тип `TOutput` не зависит от чего-либо нефиксированного, поэтому он фиксируется в `int`.
9. Теперь нефиксированных параметров типов не осталось, так что выведение успешно завершено.

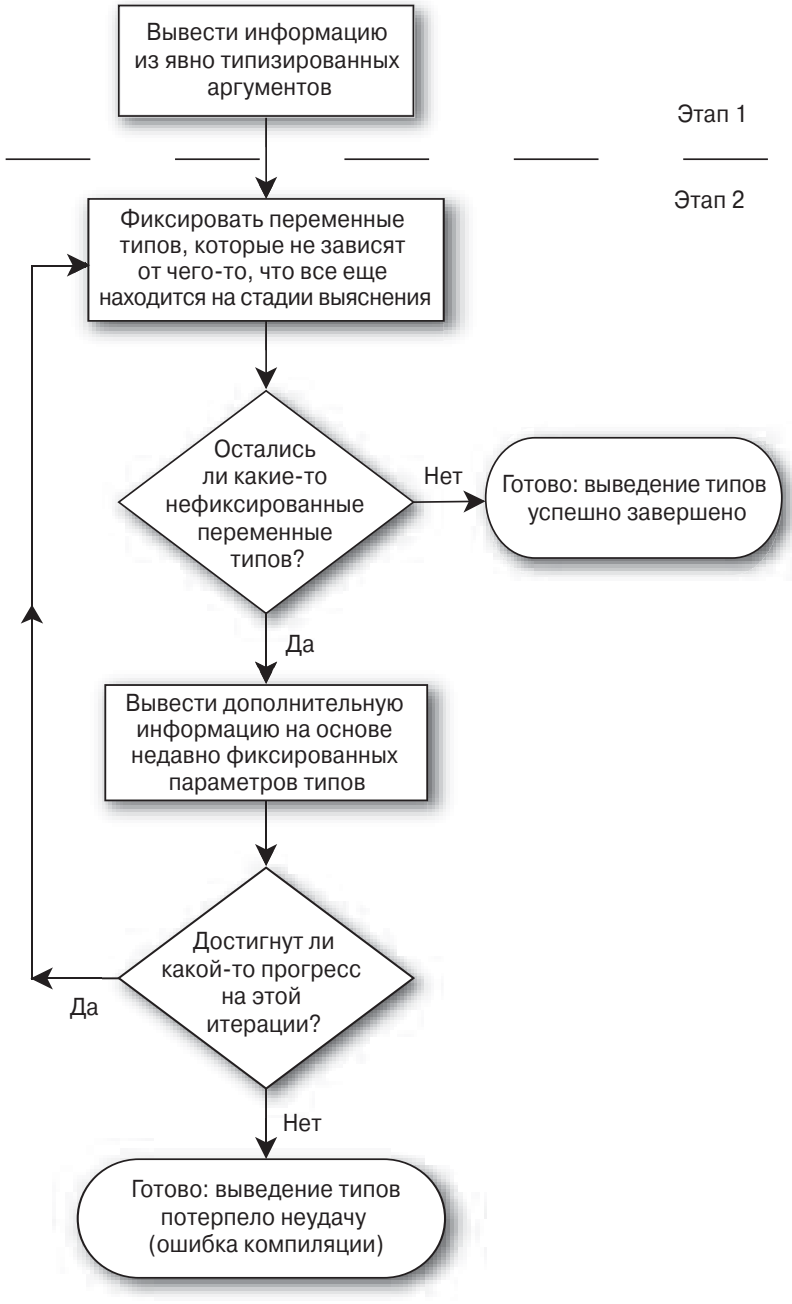

Рис. 9.7. Поток двухэтапного выведения типов

Сложно, не правда ли? Тем не менее, работа сделана — получен нужный вам результат (`TInput=string`, `TOutput=int`) и все компилируется безо всяких проблем.

Важность повторения этапа 2 лучше всего подчеркнуть с помощью еще одного примера. В листинге 9.15 демонстрируется выполнение *двух* преобразований, причем выход первого становится входом второго. До тех пор, пока не будет выяснен выходной тип первого преобразования, входной тип второго преобразования не известен, поэтому вывести его выходной тип тоже невозможно.

Листинг 9.15. Многоэтапное выведение типов

```
static void ConvertTwice<TInput,TMiddle,TOutput>
   (TInput input,
    Converter<TInput,TMiddle> firstConversion,
    Converter<TMiddle,TOutput> secondConversion)
{
   TMiddle middle = firstConversion(input);
   TOutput output = secondConversion(middle);
   Console.WriteLine(output);
}
...
ConvertTwice("Another string",
           text => text.Length,
           length => Math.Sqrt(length));
```

Первое, что следует отметить — сигнатура метода выглядит довольно устрашающе. Однако не все так плохо, если перестать бояться и взглянуть на нее внимательнее; пример применения определенно делает ее более очевидной. Здесь берется строка и над ней выполняется преобразование — то же самое преобразование, что и ранее, т.е. просто вычисление длины. Затем для этой длины строки (значение `int`) находится квадратный корень (значение `double`).

Этап 1 выведения типов сообщает компилятору о том, что должно существовать преобразование из `string` в `TInput`. Во время первого прохода этапа 2 тип `TInput` фиксируется в `string` и делается вывод о необходимости наличия преобразования из `int` в `TMiddle`. На втором проходе этапа 2 тип `TMiddle` фиксируется в `int` и делается вывод о том, что должно быть преобразование из `double` в `TOutput`. На третьем проходе этапа 2 тип `TOutput` фиксируется в `double` и выведение типов успешно завершается. По завершении выведения типов компилятор может должным образом просматривать код внутри лямбда-выражения.

Проверка тела лямбда-выражения

Тело лямбда-выражения *не может быть проверено* до тех пор, пока не станут известными типы входных параметров. Лямбда-выражение `x => x.Length` допустимо, если `x` является массивом или строкой, но недопустимо во многих других случаях. Это не проблема, когда типы параметров объявлены явно, но в случае списка неявно типизированных параметров компилятор должен подождать, пока не будет выполнено подходящее выведение типов, и только потом попытаться выяснить, в чем смысл лямбда-выражения.

Эти примеры проиллюстрировали только по одному изменению за раз, но на практике могут существовать многочисленные фрагменты информации о разных переменных типов, потенциально обнаруживаемые на различных итерациях процесса. Пытаясь спасти вашу психику (да и мою заодно), я больше не буду приводить сложные примеры, в надежде, что вы понимаете общий механизм, даже если точные детали туманны.

Хотя может показаться, что ситуация подобного рода будет возникать настолько редко, что такие сложные правила не стоит даже кратко рассматривать, на самом деле она распространена в C# 3, особенно в связи с LINQ. Вы могли бы без труда широко пользоваться выведением типов, даже не думая о нем — вполне возможно, что это войдет в вашу привычку. Но если случается отказ и вас интересует причина, вы всегда можете возвратиться к этому разделу и спецификации языка.

Необходимо ознакомиться с еще одним изменением и вас, несомненно, обрадует, что оно проще выведения типов. Давайте обратимся к перегрузке методов.

9.4.4. Выбор правильного перегруженного метода

Перегрузка предполагает наличие нескольких методов с одинаковыми именами, но разными сигнатурами. Иногда выбор метода вполне очевиден, поскольку он единственный обладает правильным количеством параметров или все аргументы в нем могут быть преобразованы в соответствующие типы параметров.

Некоторые сложности возникают в ситуации, когда подходящими *могут* быть сразу несколько методов. Правила, описанные в разделе 7.5.3 ("Overload Resolution" ("Распознавание перегруженных версий")) спецификации языка довольно сложны (да, *опять*), но основной частью является способ преобразования типов аргументов в типы параметров[9]. Например, предположим, что следующие сигнатуры методов объявлены в одном и том же типе:

```
void Write(int x)
void Write(double y)
```

Смысл вызова `Write(1.5)` очевиден, поскольку неявное преобразование из `double` в `int` отсутствует, но смысл вызова `Write(1)` определить сложнее. Неявное преобразование из `int` в `double` *существует*, поэтому оба метода допустимы. В данной точке компилятор принимает во внимание преобразование из `int` в `int` и из `int` в `double`. Преобразование типа самого в себя определяется как гораздо *лучшее*, чем преобразование в любой другой тип, так что для этого конкретного вызова метод `Write(int x)` оказывается лучше, чем `Write(double y)`.

При наличии нескольких параметров компилятор должен обеспечить применение наилучшего метода. Метод считается лучше других, если все участвующие преобразования аргументов, *по крайней мере, так же хороши*, как соответствующие преобразования в другом методе, и, по меньшей мере, одно преобразование однозначно лучше. В качестве простого примера взгляните на следующие объявления:

```
void Write(int x, double y)
void Write(double x, int y)
```

Вызов `Write(1, 1)` будет неоднозначным, и компилятор вынудит вас добавить приведение, по крайней мере, к одному параметру, чтобы прояснить, какой метод вы

[9] Предполагается, что все методы объявлены в одном классе. Когда в игру вступает наследование, все становится еще сложнее. Данный аспект в C# 3 не изменился.

намерены вызвать. Каждая перегруженная версия имеет одно лучшее преобразование аргумента, поэтому ни одна из них не может быть выбрана.

Такая логика по-прежнему применима в C# 3, но с одним дополнительным правилом, касающимся анонимных функций, в которых никогда не указывается возвращаемый тип. В этом случае в правилах наилучшего преобразования используется выведенный возвращаемый тип (как описано в разделе 9.4.2).

Давайте рассмотрим пример ситуации, когда требуется это новое правило. В листинге 9.16 содержится два метода по имени `Execute()` и вызов, в котором применяется лямбда-выражение.

Листинг 9.16. Пример выбора перегруженной версии при влиянии возвращаемого типа делегата

```
static void Execute(Func<int> action)
{
  Console.WriteLine("action returns an int: " + action());
}
static void Execute(Func<double> action)
{
  Console.WriteLine("action returns a double: " + action());
}
...
Execute(() => 1);
```

Вызов `Execute()` в листинге 9.16 взамен можно было бы записать с помощью анонимного метода или группы методов — какой бы вид преобразования не был задействован, применяются те же самые правила. Какой из методов `Execute()` должен быть вызван? Правила перегрузки гласят, что когда после преобразований аргументов оба метода применимы, эти преобразования аргументов анализируются на предмет выявления лучшего из них. Преобразования здесь осуществляются не из обычного типа .NET в тип параметра, а из лямбда-выражения в два типа делегатов. Какое из преобразований лучше?

Как ни удивительно, но эта же ситуация в C# 2 привела бы к ошибке компиляции — языковые правила для такого случая не предусмотрены. В C# 3 будет выбран метод с параметром `Func<int>`. Добавленное дополнительное правило может быть изложено своими словами следующим образом.

Если анонимная функция может быть преобразована в два типа делегатов, которые имеют одинаковые списки параметров, но отличающиеся возвращаемые типы, то преобразования делегатов оцениваются по преобразованиям из выведенного возвращаемого типа в возвращаемые типы делегатов.

Без ссылки на пример звучит как порядочная тарабарщина. Давайте снова возвратимся к листингу 9.16, в котором выполнялось преобразование из лямбда-выражения, не принимающего параметров и имеющего выведенный возвращаемый тип `int`, либо в тип `Func<int>`, либо в тип `Func<double>`. Для обоих типов делегатов списки параметров одинаковы (пустые), поэтому применяется указанное выше правило. Затем нужно просто найти лучшее преобразование: `int` в `int` или `int` в `double`. Ситуация должна выглядеть знакомой; как было упомянуто ранее, преобразование `int` в `int` считается лучшим. Таким образом, код из листинга 9.16 выводит на консоль строку `action returns an int: 1`.

9.4.5. Итоги по выведению типов и распознаванию перегруженных версий

Материал этого раздела был довольно трудным. Я хотел бы сделать его проще, но он был посвящен фундаментально сложной теме. Задействованная терминология не способствует упрощению, да еще и с учетом того, что *тип параметра* и *параметр типа* обозначают совершенно разные вещи! Примите поздравления, если вы дошли до конца и действительно все поняли. Не переживайте, если это не так; надеюсь, что когда вы будете читать данный раздел в следующий раз, он прольет больше света на эту тему — особенно после того, как вы попали в ситуацию, непосредственно касающуюся рассматриваемых здесь вопросов. Ниже перечислены наиболее важные моменты, накопившиеся к этому времени.

- Анонимные функции (анонимные методы и лямбда-выражения) имеют выведенные возвращаемые типы, основанные на типах, которые использовались во всех операторах `return`.
- Лямбда-выражения могут восприниматься компилятором, только когда известны типы всех их параметров.
- Выведение типов больше не требует, чтобы каждый аргумент независимо приходил к точно такому же заключению относительно параметров типов, при условии совместимости результатов.
- Выведение типов теперь является многоэтапным: выведенный возвращаемый тип одной анонимной функции может выступать в качестве типа параметра для другой такой функции.
- При поиске лучшей перегруженной версии метода, когда участвуют анонимные функции, принимается во внимание возвращаемый тип.

Даже такой короткий список очень плотно усеян техническими терминами. Не волнуйтесь, если вам не все в нем понятно. По моему опыту в большинстве случаев все работает так, как нужно.

9.5. Резюме

В C# 3 лямбда-выражения почти полностью заменили анонимные методы. Анонимные методы поддерживаются ради обратной совместимости, но идиоматический, заново написанный код C# 3 будет содержать их мало.

Вы видели, что лямбда-выражения — это нечто большее, чем просто компактный синтаксис для создания делегатов. Они могут быть преобразованы в деревья выражений с учетом ряда ограничений. Деревья выражений затем могут обрабатываться другим кодом, возможно выполняющим эквивалентные действия в разных исполняющих средах. Без такой характеристики язык LINQ ограничивался бы внутрипроцессными запросами.

Наше обсуждение выведения типов было в определенной степени необходимым злом; лишь немногим разработчикам действительно *нравится* говорить о такой разновидности правил, которые при этом должны применяться, но важно иметь хотя бы примерное представление о том, что происходит. Прежде чем начать чрезмерно жалеть самих себя, подумайте о бедных проектировщиках языка, которым приходилось жить и дышать этим, удостоверяясь, что правила согласованы и не разваливаются в неприятных ситуациях. Затем вспомните о тестировщиках, которые должны были пытаться нарушить работу реализации. С точки зрения *описания* лямбда-выражений на этом все, но вы еще увидите много случаев их применения в остальных главах книги. Например, в следующей главе подробно рассматриваются *расширяющие методы*. На первый взгляд, они полностью отделены от лямбда-выражений, но на деле эти два средства часто используются вместе.