

`IComparable`, не перегружают операции `>` и `<`. Это отличается от ситуации с эквивалентностью, при которой в случае переопределения метода `Equals` обычно перегружается операция `==`.

Как правило, операции `>` и `<` перегружаются только в перечисленных ниже случаях.

- Тип обладает строгой внутренне присущей концепцией “больше чем” и “меньше чем” (в противоположность более широким концепциям “находится перед” и “находится после”, принятым в интерфейсе `IComparable`).
- Существует только один способ или контекст, в котором производится сравнение.
- Результат инвариантен для различных культур.

Класс `System.String` не удовлетворяет последнему пункту: результаты сравнения строк в разных языках могут варьироваться. Следовательно, тип `string` не поддерживает операции `>` и `<`:

```
bool error = "Beck" > "Anne"; // Ошибка на этапе компиляции
```

Реализация интерфейсов `IComparable`

В следующей структуре, представляющей музыкальную ноту, мы реализуем интерфейсы `IComparable`, а также перегружаем операции `<` и `>`. Для полноты мы также переопределяем методы `Equals/GetHashCode` и перегружаем операции `==` и `!=`:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
    int _semitonesFromA;
    public int SemitonesFromA { get { return _semitonesFromA; } }
    public Note (int semitonesFromA)
    {
        _semitonesFromA = semitonesFromA;
    }
    public int CompareTo (Note other) // Обобщенный интерфейс IComparable<T>
    {
        if (Equals (other)) return 0; // Отказоустойчивая проверка
        return _semitonesFromA.CompareTo (other._semitonesFromA);
    }
    int IComparable.CompareTo (object other) // Необобщенный интерфейс IComparable
    {
        if (!(other is Note))
            throw new InvalidOperationException ("CompareTo: Not a note");
        // не нота
        return CompareTo ((Note) other);
    }
    public static bool operator < (Note n1, Note n2)
        => n1.CompareTo (n2) < 0;
    public static bool operator > (Note n1, Note n2)
        => n1.CompareTo (n2) > 0;
```

```

public bool Equals (Note other) // для IEquatable<Note>
    => _semitonesFromA == other._semitonesFromA;
public override bool Equals (object other)
{
    if (!(other is Note)) return false;
    return Equals ((Note) other);
}
public override int GetHashCode () => _semitonesFromA.GetHashCode ();
// Вызвать статический метод Equals, чтобы гарантировать
// надлежащую обработку значений null:
public static bool operator == (Note n1, Note n2) => Equals (n1, n2);
public static bool operator != (Note n1, Note n2) => !(n1 == n2);
}

```

Служебные классы

Класс Console

Статический класс `Console` обрабатывает ввод-вывод для консольных приложений. В приложении командной строки (консольном приложении) ввод поступает с клавиатуры через методы `Read`, `ReadKey` и `ReadLine`, а вывод осуществляется в текстовое окно посредством методов `Write` и `WriteLine`. Управлять позицией и размерами такого окна можно с помощью свойств `WindowLeft`, `WindowTop`, `WindowHeight` и `WindowWidth`. Можно также изменять свойства `BackgroundColor` и `ForegroundColor` и манипулировать курсором через свойства `CursorLeft`, `CursorTop` и `CursorSize`:

```

Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%"); // test... 90%

```

Методы `Write` и `WriteLine` перегружены для приема смешанной форматной строки (см. раздел “Метод `String.Format` и смешанные форматные строки” ранее в главе). Тем не менее, ни один из методов не принимает поставщик формата, так что вы привязаны к `CultureInfo.CurrentCulture`. (Разумеется, существует обходной путь, предусматривающий явный вызов `string.Format`.)

Свойство `Console.Out` возвращает объект `TextWriter`. Передача `Console.Out` методу, который ожидает `TextWriter` — удобный способ заставить этот метод выводить что-либо на консоль в целях диагностики.

Можно также перенаправлять потоки ввода и вывода на консоль с помощью методов `SetIn` и `SetOut`:

```

// Сначала сохранить существующий объект записи вывода:
System.IO.TextWriter oldOut = Console.Out;

// Перенаправить консольный вывод в файл:
using (System.IO.TextWriter w = System.IO.File.CreateText
        ("e:\\output.txt"))

```

```
{  
    Console.SetOut (w);  
    Console.WriteLine ("Hello world");  
}  
// Восстановить стандартный консольный вывод:  
Console.SetOut (oldOut);
```

Работа потоков данных и объектов записи текста обсуждается в главе 15.



При запуске приложений WPF или Windows Forms в среде Visual Studio консольный вывод автоматически перенаправляется в окно вывода Visual Studio (в режиме отладки). Такая особенность делает метод `Console.WriteLine` полезным для диагностических целей, хотя в большинстве случаев более подходящими будут классы `Debug` и `Trace` из пространства имен `System.Diagnostics` (см. главу 13).

Класс `Environment`

Статический класс `System.Environment` предлагает набор полезных свойств, предназначенных для взаимодействия с разнообразными сущностями.

Файлы и папки

`CurrentDirectory`, `SystemDirectory`, `CommandLine`

Компьютер и операционная система

`MachineName`, `ProcessorCount`, `OSVersion`, `NewLine`

Пользователь, вошедший в систему

`UserName`, `UserInteractive`, `UserDomainName`

Диагностика

`TickCount`, `StackTrace`, `WorkingSet`, `Version`

Дополнительные папки можно получить вызовом метода `GetFolderPath`; мы рассмотрим его в разделе “Операции с файлами и каталогами” главы 15.

Обращаться к переменным среды операционной системы (которые просматриваются за счет ввода `set` в командной строке) можно с помощью следующих трех методов: `GetEnvironmentVariable`, `GetEnvironmentVariables` и `SetEnvironmentVariable`.

Свойство `ExitCode` позволяет установить код возврата, предназначенный для ситуации, когда программа запускается из команды либо из пакетного файла, а метод `FailFast` завершает программу немедленно, не выполняя очистку.

Класс `Environment`, доступный приложениям из Магазина Microsoft, предлагает только ограниченное количество членов (`ProcessorCount`, `NewLine` и `FailFast`).

Класс `Process`

Класс `Process` из пространства имен `System.Diagnostics` позволяет запускать новый процесс. (В главе 13 будет описано, как его можно применять также для взаимодействия с другими процессами, выполняющимися на компьютере.)



По причинам, связанным с безопасностью, класс `Process` не доступен приложениям из Магазина Microsoft, поэтому запускать произвольные процессы невозможно. Взамен должен использоваться класс `Windows.System.Launcher` для “запуска” URI или файла, к которому имеется доступ, например:

```
Launcher.LaunchUriAsync (new Uri ("http://albahari.com"));
var file = await KnownFolders.DocumentsLibrary.GetFileAsync("foo.txt");
Launcher.LaunchFileAsync (file);
```

Такой код приводит к открытию URI или файла с применением любой программы, ассоциированной со схемой URI или файловым расширением. Чтобы все работало, программа должна функционировать на переднем плане.

Статический метод `Process.Start` имеет несколько перегруженных версий; простейшая из них принимает имя файла с необязательными аргументами:

```
Process.Start ("notepad.exe");
Process.Start ("notepad.exe", "e:\\file.txt");
```

Наиболее гибкая перегруженная версия принимает экземпляр `ProcessStartInfo`. С его помощью можно захватывать и перенаправлять потоки ввода, вывода и ошибок запущенного процесса (если свойство `UseShellExecute` установлено в `false`). В следующем коде захватывается вывод утилиты ipconfig:

```
ProcessStartInfo psi = new ProcessStartInfo
{
    FileName = "cmd.exe",
    Arguments = "/c ipconfig /all",
    RedirectStandardOutput = true,
    UseShellExecute = false
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```

Если вывод не перенаправлен, тогда метод `Process.Start` выполняет программу параллельно с вызывающей программой. Если нужно ожидать завершения нового процесса, то можно вызвать метод `WaitForExit` на объекте `Process` с указанием необязательного тайм-аута.

Перенаправление потоков вывода и ошибок

Когда свойство `UseShellExecute` установлено в `false` (как принято по умолчанию в .NET), вы можете захватывать стандартные потоки ввода, вывода и ошибок, после чего производить с ними записи/чтение через свойства `StandardInput`, `StandardOutput` и `StandardError`.

Сложность возникает при необходимости перенаправления и стандартного потока вывода, и стандартного потока ошибок, поскольку обычно вам не будет известно, в каком порядке читать данные из каждого потока (т.к. невозможно знать заранее, каким образом чередуются данные). Решение заключается в том, чтобы читать сразу оба потока, чего можно добиться, читая (хотя бы) из одного потока *асинхронно*. Ниже описано, как нужно поступать.

- Предусмотрите обработку событий `OutputDataReceived` и/или `ErrorDataReceived`, которые инициируются при получении данных вывода/ошибки.
- Вызовите метод `BeginOutputReadLine` и/или `BeginErrorReadLine`, что включает упомянутые выше события.

Следующий метод запускает исполняемый модуль, одновременно захватывая потоки вывода и ошибок:

```
(string output, string errors) Run (string exePath, string args = "")  
{  
    using var p = Process.Start (new ProcessStartInfo (exePath, args)  
    {  
        RedirectStandardOutput = true,  
        RedirectStandardError = true, UseShellExecute = false,  
    });  
    var errors = new StringBuilder ();  
    // Читать из потока ошибок асинхронно...  
    p.ErrorDataReceived += (sender, errorArgs) =>  
    {  
        if (errorArgs.Data != null) errors.AppendLine (errorArgs.Data);  
    };  
    p.BeginErrorReadLine ();  
    // ...наряду с чтением потока вывода синхронно:  
    string output = p.StandardOutput.ReadToEnd();  
    p.WaitForExit();  
    return (output, errors.ToString());  
}
```

Флаг `UseShellExecute`



В .NET 5+ (и .NET Core) флаг `UseShellExecute` по умолчанию установлен в `false`, тогда как в .NET Framework его стандартным значением было `true`. Поскольку это критическое изменение, при переносе кода из .NET Framework стоит проверить все вызовы метода `Process.Start`.

Флаг `UseShellExecute` изменяет способ запуска процесса средой CLR. Когда свойство `UseShellExecute` установлено в `true`, вы можете выполнять перечисленные ниже действия.

- Указывать путь к файлу или документу, а не к исполняемому модулю (в результате чего операционная система открывает файл или документ с помощью ассоциированного приложения).
- Указывать URL (в результате чего операционная система перейдет по этому URL в стандартном веб-браузере).
- (Только в Windows.) Указывать команду (такую как `runas` для запуска процесса с правами, повышенными до администратора).

Недостаток заключается в том, что перенаправление потоков ввода или вывода невозможно. Если вас интересует перенаправление при запуске файла или документа, то существует обходной способ — установите `UseShellExecute` в

`false` и вызовите процесс командной строки (`cmd.exe`) с переключателем `/c`, как делалось ранее в отношении вызова `ipconfig`.

В среде Windows флаг `UseShellExecute` инструктирует CLR о необходимости использования Windows-функции `ShellExecute`, а не `CreateProcess`. В среде Linux флаг `UseShellExecute` указывает CLR о том, что нужно вызвать `xdg-open`, `gnome-open` или `kfmclient`.

Класс `AppContext`

Статический класс `System.AppContext` предлагает два полезных свойства.

- `BaseDirectory` возвращает папку, в которой было запущено приложение. Такая папка важна для распознавания сборок (поиска и загрузки зависимостей) и нахождения конфигурационных файлов (таких как `appsettings.json`).
- `TargetFrameworkName` сообщает имя и версию исполняющей среды .NET, на которую нацелено приложение (как указано в его файле `.runtimeconfig.json`). Она может быть более старой, чем фактически используемая исполняющая среда.

В добавок класс `System.AppContext` ведет глобальный словарь булевых значений со строковыми ключами, предназначенный для разработчиков библиотек в качестве стандартного механизма, который позволяет потребителям включать и отключать новые функциональные средства. Подобный нетипизированный подход имеет смысл применять с экспериментальными функциональными средствами, которые желательно сохранять недокументированными для большинства пользователей.

Потребитель библиотеки требует, чтобы определенное функциональное средство было включено, следующим образом:

```
AppContext.SetSwitch ("MyLibrary.SomeBreakingChange", true);
```

Затем код внутри библиотеки может проверять признак включения функционального средства:

```
bool.isDefined, switchValue;
isDefined = AppContext.TryGetSwitch ("MyLibrary.SomeBreakingChange",
                                    out switchValue);
```

Метод `TryGetSwitch` возвращает `false`, если признак включения не определен; это позволяет различать неопределенный признак включения от призыва, значение которого установлено в `false`, когда в таком действии возникнет необходимость.



По иронии судьбы проектное решение, положенное в основу метода `TryGetSwitch`, иллюстрирует то, как не следует разрабатывать API-интерфейсы. Параметр `out` является излишним, а взамен метод должен возвращать допускающий `null` тип `bool`, который принимает значение `true`, `false` или же `null` для неопределенного призыва включения. В таком случае вот как его можно было бы использовать:

```
bool switchValue = AppContext.GetSwitch ("...") ?? false;
```




Коллекции

В .NET имеется стандартный набор типов для хранения и управления коллекциями объектов. В него входят списки с изменяемыми размерами, связные списки, отсортированные и несортированные словари и массивы. Из всего перечисленного только массивы являются частью языка C#, остальные коллекции представляют собой просто классы, экземпляры которых можно создавать подобно любым другим классам.

Типы в библиотеке .NET BCL для коллекций могут быть разделены на следующие категории:

- интерфейсы, которые определяют стандартные протоколы коллекций;
- готовые к использованию классы коллекций (списки, словари и т.д.);
- базовые классы, которые позволяют создавать коллекции, специфичные для приложений.

В настоящей главе рассматриваются все указанные категории, а также типы, применяемые при определении эквивалентности и порядка следования элементов. Ниже перечислены пространства имен коллекций.

Пространство имен	Что содержит
System.Collections	Необщенные классы и интерфейсы коллекций
System.Collections.Specialized	Строго типизированные необщенные классы коллекций
System.Collections.Generic	Обобщенные классы и интерфейсы коллекций
System.Collections.ObjectModel	Посредники и базовые классы для специальных коллекций
System.Collections.Concurrent	Коллекции, безопасные к потокам (глава 22)

Перечисление

В программировании существует много разнообразных коллекций, начиная с простых структур данных вроде массивов и связных списков и заканчивая сложными структурами, такими как красно-черные деревья и хеш-таблицы. Хотя внутренняя реализация и внешние характеристики таких структур данных варьируются в широких пределах, наиболее универсальной потребностью является возможность прохода по содержимому коллекции. Библиотека .NET BCL поддерживает эту потребность через пару интерфейсов (`IEnumerable`, `IEnumerator` и их обобщенные аналоги), которые позволяют различным структурам данных открывать доступ к общим API-интерфейсам обхода. Они представляют собой часть более крупного множества интерфейсов коллекций, которые показаны на рис. 7.1.

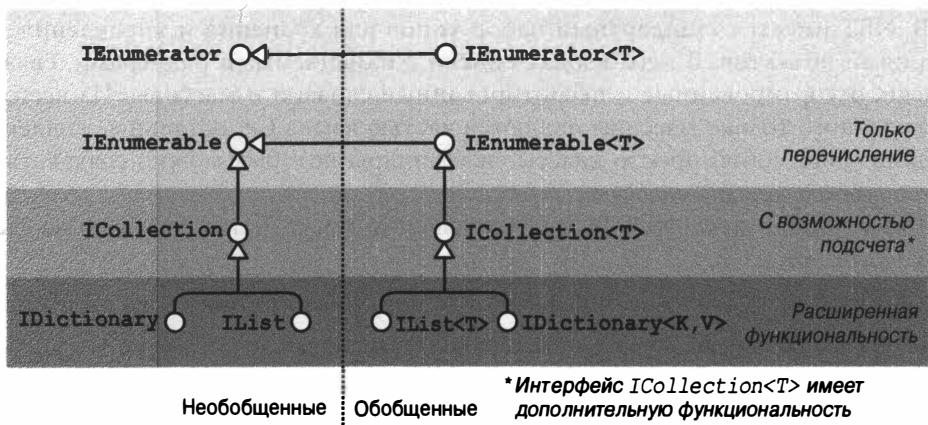


Рис. 7.1. Интерфейсы коллекций

Интерфейсы `IEnumerable` и `IEnumerator`

Интерфейс `IEnumerator` определяет базовый низкоуровневый протокол, посредством которого производится проход по элементам — или перечисление — коллекции в одностороннем стиле. Объявление этого интерфейса показано ниже:

```
public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Метод `MoveNext` продвигает текущий элемент, или “курсор”, на следующую позицию, возвращая `false`, если в коллекции больше не осталось элементов. Метод `Current` возвращает элемент в текущей позиции (обычно приводя его от `object` к более специальному типу). Перед извлечением первого элемента должен быть вызван метод `MoveNext`, что нужно для учета пустой коллекции.

Метод `Reset`, когда он реализован, выполняет перемещение в начало, делая возможным перечисление коллекции заново. Метод `Reset` существует главным образом для взаимодействия с СОМ: вызова его напрямую в общем случае избегают, т.к. он не является универсально поддерживаемым (и он необязателен, потому что обычно просто создает новый экземпляр перечислителя).

Как правило, коллекции не *реализуют* перечислители; взамен они *предоставляют* их через интерфейс `IEnumerable`:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

За счет определения единственного метода, возвращающего перечислитель, интерфейс `IEnumerable` обеспечивает гибкость в том, что реализация логики итерации может быть возложена на другой класс. Кроме того, это означает, что несколько потребителей способны выполнять перечисление коллекции одновременно, не влияя друг на друга. Интерфейс `IEnumerable` можно воспринимать как “поставщик `IEnumerator`”, и он является наиболее базовым интерфейсом, который реализуют классы коллекций.

В следующем примере демонстрируется низкоуровневое использование интерфейсов `IEnumerable` и `IEnumerator`:

```
string s = "Hello";
// Так как тип string реализует IEnumerable, мы можем вызывать GetEnumerator():
IEnumerator rator = s.GetEnumerator();
while (rator.MoveNext())
{
    char c = (char) rator.Current;
    Console.Write (c + ".");
}
// Вывод: H.e.l.l.o.
```

Однако вызов методов перечислителей напрямую в такой манере встречается редко, поскольку C# предоставляет синтаксическое сокращение: оператор `foreach`. Ниже приведен тот же самый пример, переписанный с применением `foreach`:

```
string s = "Hello"; // Класс String реализует интерфейс IEnumerable
foreach (char c in s)
    Console.Write (c + ".");
```

Интерфейсы `IEnumerable<T>` и `IEnumerator<T>`

Интерфейсы `IEnumerator` и `IEnumerable` почти всегда реализуются в сочетании со своими расширенными обобщенными версиями:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
    T Current { get; }
}
```

```
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Определяя типизированные версии свойства `Current` и метода `GetEnumerator`, данные интерфейсы усиливают статическую безопасность типов, избегают накладных расходов на упаковку элементов типов значений и повышают удобство эксплуатации потребителями. Массивы автоматически реализуют интерфейс `IEnumerable<T>` (где `T` — тип элементов массива).

Благодаря улучшенной статической безопасности типов вызов следующего метода с массивом символов приведет к возникновению ошибки на этапе компиляции:

```
void Test (IEnumerable<int> numbers) { ... }
```

Для классов коллекций стандартной практикой является открытие общего доступа к `IEnumerable<T>` и в то же время “сокрытие” необобщенного `IEnumerable` посредством явной реализации интерфейса. Это значит, что в случае вызова напрямую метода `GetEnumerator` будет получена реализация безопасного к типам обобщенного интерфейса `IEnumerator<T>`. Тем не менее, временами такое правило нарушается из-за необходимости обеспечения обратной совместимости (до выхода версии C# 2.0 обобщения не существовали). Хорошим примером считаются массивы — во избежание нарушения работы ранее написанного кода они должны возвращать экземпляр реализации необобщенного (можно сказать “классического”) интерфейса `IEnumerator`. Для того чтобы получить обобщенный интерфейс `IEnumerable<T>`, придется выполнить явное приведение к соответствующему интерфейсу:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

К счастью, благодаря наличию оператора `foreach` писать код подобного рода приходится редко.

Интерфейсы `IEnumerable<T>` и `IDisposable`

Интерфейс `IEnumerable<T>` унаследован от `IDisposable`. Это позволяет перечислителям хранить ссылки на такие ресурсы, как подключения к базам данных, и гарантировать, что ресурсы будут освобождены, когда перечисление завершится (или прекратится на полпути). Оператор `foreach` распознает такие детали и транслирует следующий код:

```
foreach (var element in somethingEnumerable) { ... }
```

в его логический эквивалент:

```
using (var rator = somethingEnumerable.GetEnumerator())
{
    while (rator.MoveNext())
    {
        var element = rator.Current;
        ...
    }
}
```

Блок `using` обеспечивает освобождение (более подробно об интерфейсе `IDisposable` речь пойдет в главе 12).

Когда необходимо использовать необобщенные интерфейсы?

С учетом дополнительной безопасности типов, присущей обобщенным интерфейсам коллекций вроде `IEnumerable<T>`, возникает вопрос: нужно ли вообще применять необобщенный интерфейс `IEnumerable` (либо `ICollection` или `IList`)?

В случае интерфейса `IEnumerable` вы должны реализовать его в сочетании с `IEnumerable<T>`, т.к. последний является производным от первого. Однако вы очень редко будете действительно реализовывать данные интерфейсы с нуля: почти во всех ситуациях можно принять высокоуровневый подход, предусматривающий использование методов итератора, `Collection<T>` и LINQ.

А что насчет потребителя? Практически во всех случаях управление может осуществляться целиком с помощью обобщенных интерфейсов. Тем не менее, необобщенные интерфейсы по-прежнему иногда полезны своей способностью обеспечивать унификацию типов для коллекций по всем типам элементов. Например, показанный ниже метод подсчитывает элементы в любой коллекции *рекурсивным образом*:

```
public static int Count (IEnumerable e)
{
    int count = 0;
    foreach (object element in e)
    {
        var subCollection = element as IEnumerable;
        if (subCollection != null)
            count += Count (subCollection);
        else
            count++;
    }
    return count;
}
```

Поскольку язык C# предлагает ковариантность с обобщенными интерфейсами, может показаться допустимым заставить данный метод взамен принимать тип `IEnumerable<object>`. Однако тогда метод потерпит неудачу с элементами типов значений и унаследованными коллекциями, которые не реализуют интерфейс `IEnumerable<T>` — примером может служить класс `ControlCollection` из Windows Forms.

(Кстати, в приведенном примере вы могли заметить потенциальную ошибку: циклические ссылки приведут к бесконечной рекурсии и аварийному отказу метода. Устранить проблему проще всего за счет применения типа `HashSet` — см. раздел “Классы `HashSet<T>` и `SortedSet<T>`” далее в главе.)

Реализация интерфейсов перечисления

Реализация интерфейса `IEnumerable` или `IEnumerable<T>` может понадобиться по одной или нескольким описанным ниже причинам:

- для поддержки оператора `foreach`;
- для взаимодействия со всем, что ожидает стандартной коллекции;
- для удовлетворения требований более развитого интерфейса коллекции;
- для поддержки инициализаторов коллекций.

Чтобы реализовать `IEnumerable/IEnumerable<T>`, потребуется предоставить перечислитель. Это можно сделать одним из трех способов:

- если класс является оболочкой для другой коллекции, то путем возвращения перечислителя внутренней коллекции;
- через итератор с использованием оператора `yield return`;
- за счет создания экземпляра собственной реализации `IEnumerator/IEnumerator<T>`.



Можно также создать подкласс существующей коллекции: класс `Collection<T>` предназначен как раз для такой цели (см. раздел “Настраиваемые коллекции и посредники” далее в главе). Еще один подход предусматривает применение операций запросов LINQ, которые рассматриваются в следующей главе.

Возвращение перечислителя другой коллекции сводится к вызову метода `GetEnumerator` на внутренней коллекции. Однако данный прием жизнеспособен только в простейших сценариях, где элементы во внутренней коллекции являются в точности тем, что требуется. Более гибкий подход заключается в написании итератора с использованием оператора `yield return`. Итератор — это средство языка C#, которое помогает в написании коллекций таким же способом, каким оператор `foreach` оказывает содействие в их потреблении. Итератор автоматически поддерживает реализацию интерфейсов `IEnumerable` и `IEnumerator` либо их обобщенных версий. Ниже приведен простой пример:

```
public class MyCollection : IEnumerable
{
    int[] data = { 1, 2, 3 };

    public IEnumerator GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }
}
```

Обратите внимание на “черную магию”: кажется, что метод `GetEnumerator` вообще не возвращает перечислитель! Столкнувшись с оператором `yield return`, компилятор “за кулисами” генерирует скрытый вложенный класс перечислителя, после чего проводит рефакторинг метода `GetEnumerator` для созда-

ния и возвращения экземпляра данного класса. Итераторы отличаются мощью и простотой (и широко применяются в реализации стандартных операций запросов LINQ to Objects).

Придерживаясь такого подхода, мы можем также реализовать обобщенный интерфейс `IEnumerable<T>`:

```
public class MyGenCollection : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    public IEnumerator<int> GetEnumerator()
    {
        foreach (int i in data) yield return i;
    }

    // Явная реализация сохраняет его скрытым:
    IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Из-за того, что интерфейс `IEnumerable<T>` унаследован от `IEnumerable`, мы должны реализовывать как обобщенную, так и необобщенную версию метода `GetEnumerator`. В соответствии со стандартной практикой мы реализовали необобщенную версию явно. Она может просто вызывать обобщенную версию `GetEnumerator`, поскольку интерфейс `IEnumerable<T>` унаследован от `IEnumerable`.

Только что написанный класс подошел бы в качестве основы для создания более развитой коллекции. Тем не менее, если ничего сверх простой реализации интерфейса `IEnumerable<T>` не требуется, то оператор `yield return` предлагает упрощенный вариант. Вместо написания класса логику итерации можно переместить внутрь метода, возвращающего обобщенную реализацию `IEnumerable<T>`, и позволить компилятору позаботиться об остальном. Например:

```
public class Test
{
    public static IEnumerable <int> GetSomeIntegers()
    {
        yield return 1;
        yield return 2;
        yield return 3;
    }
}
```

А вот как использовать такой метод:

```
foreach (int i in Test.GetSomeIntegers())
    Console.WriteLine (i);
```

Последний подход к написанию метода `GetEnumerator` предполагает построение класса, который реализует интерфейс `IEnumerable` напрямую. Это в точности то, что компилятор делает “за кулисами”, когда распознает итераторы. (К счастью, заходить настолько далеко вам придется редко.) В следующем примере определяется коллекция, содержащая целые числа 1, 2 и 3:

```

public class MyIntList : IEnumerable
{
    int[] data = { 1, 2, 3 };

    public IEnumerator GetEnumerator() => new Enumerator (this);

    class Enumerator : IEnumerator
    {
        MyIntList collection; // Определить внутренний
        int currentIndex = -1; // класс для перечислителя

        public Enumerator (MyIntList items) => this.collection = items;

        public object Current
        {
            get
            {
                if (currentIndex == -1)
                    throw new InvalidOperationException ("Enumeration not started!");
                    // Перечисление не началось!
                if (currentIndex == collection.data.Length)
                    throw new InvalidOperationException ("Past end of list!");
                    // Пройден конец списка!
                return collection.data [currentIndex];
            }
        }

        public bool MoveNext()
        {
            if (currentIndex >= collection.data.Length - 1) return false;
            return ++currentIndex < collection.data.Length;
        }

        public void Reset() => currentIndex = -1;
    }
}

```



Реализовывать метод `Reset` вовсе не обязательно — взамен можно сгенерировать исключение `NotSupportedException`.

Обратите внимание, что первый вызов `MoveNext` должен переместить на первый (а не на второй) элемент в списке.

Чтобы сравняться с итератором в плане функциональности, мы должны также реализовать интерфейс `IEnumerable<T>`. Ниже приведен пример, в котором для простоты опущена проверка границ:

```

class MyIntList : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    // Обобщенный перечислитель совместим как
    // с IEnumerable, так и с IEnumerable<T>.
    // Во избежание конфликта имен мы реализуем
    // необобщенный метод GetEnumerator явно.

    public IEnumerator<int> GetEnumerator() => new Enumerator(this);
    IEnumerable GetEnumerator() => new Enumerator(this);

```

```

class Enumerator : IEnumerator<int>
{
    int currentIndex = -1;
    MyIntList collection;

    public Enumerator (MyIntList items) => collection = items;
    public int Current => collection.data [currentIndex];
    object IEnumerator.Current => Current;

    public bool MoveNext() => ++currentIndex < collection.data.Length;
    public void Reset() => currentIndex = -1;
    // С учетом того, что метод Dispose не нужен,
    // рекомендуется реализовать его явно, чтобы
    // он не был виден через открытый интерфейс.
    void IDisposable.Dispose() {}
}
}

```

Пример с обобщениями функционирует быстрее, т.к. свойство `IEnumerator<int>.Current` не требует приведения `int` к `object`, что позволяет избежать накладных расходов, связанных с упаковкой.

Интерфейсы `ICollection` и `IList`

Хотя интерфейсы перечислений предлагают протокол для односторонней итерации по коллекции, они не предоставляют механизма, который позволил бы определять размер коллекции, получать доступ к членам по индексу, производить поиск или модифицировать коллекцию. Для такой функциональности в .NET определены интерфейсы `ICollection`, `IList` и `IDictionary`. Каждый из них имеет обобщенную и необобщенную версии; тем не менее, необобщенные версии существуют преимущественно для поддержки унаследованного кода.

Иерархия наследования для этих интерфейсов была показана на рис. 7.1. Резюмировать их проще всего следующим образом.

`IEnumerable<T>` (и `IEnumerable`)

Предоставляют минимальный уровень функциональности (только перечисление).

`ICollection<T>` (и `ICollection`)

Предоставляют средний уровень функциональности (например, свойство `Count`).

`IList<T>/IDictionary<K, V>` и их необобщенные версии

Предоставляют максимальный уровень функциональности (включая “произвольный” доступ по индексу/ключу).



Потребность в реализации любого из упомянутых интерфейсов возникает редко. Когда необходимо написать класс коллекции, почти во всех случаях можно создавать подкласс класса `Collection<T>` (см. раздел “Настраиваемые коллекции и посредники” далее в главе). Язык LINQ предлагает еще один вариант, охватывающий множество сценариев.

Обобщенная и необобщенная версии отличаются между собой сильнее, чем можно было бы ожидать, особенно в случае интерфейса `ICollection`. Причины по большей части исторические: поскольку обобщения появились позже, обобщенные интерфейсы были разработаны с оглядкой на прошлое, что привело к отличающемуся (и лучшему) подбору членов. В итоге интерфейс `ICollection<T>` не расширяет `ICollection`, `IList<T>` не расширяет `IList`, а `IDictionary< TKey, TValue >` не расширяет `IDictionary`. Разумеется, сам класс коллекции свободен в реализации обеих версий интерфейса, если это приносит пользу (часто именно так и есть).



Другая, более тонкая причина того, что интерфейс `IList<T>` не расширяет `IList`, заключается в том, что тогда приведение к `IList<T>` возвращало бы реализацию интерфейса с членами `Add(T)` и `Add(object)`. В результате нарушилась бы статическая безопасность типов, потому что метод `Add` можно было бы вызывать с объектом любого типа.

В текущем разделе рассматриваются интерфейсы `ICollection<T>` и `IList<T>` плюс их необобщенные версии, а в разделе “Словари” далее в главе обсуждаются интерфейсы словарей.



Не существует *разумного* объяснения способа применения слов *коллекция* и *список* повсюду в библиотеках .NET. Например, поскольку `IList<T>` является более функциональной версией `ICollection<T>`, можно было бы ожидать, что класс `List<T>` должен быть соответственно более функциональным, чем класс `Collection<T>`. Но это не так. Лучше всего считать термины *коллекция* и *список* в широком смысле синонимами кроме случаев, когда речь идет о конкретном типе.

Интерфейсы `ICollection<T>` и `ICollection`

`ICollection<T>` — стандартный интерфейс для коллекций объектов с поддержкой подсчета. Он предоставляет возможность определения размера коллекции (`Count`), выяснения, содержит ли элемент в коллекции (`Contains`), копирования коллекции в массив (`ToArray`) и определения, предназначена ли коллекция только для чтения (`IsReadOnly`). Для записываемых коллекций можно также добавлять (`Add`), удалять (`Remove`) и очищать (`Clear`) элементы коллекции. Из-за того, что интерфейс `ICollection<T>` расширяет `IEnumerable<T>`, можно также совершать обход с помощью оператора `foreach`:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }

    bool Contains (T item);
    void CopyTo (T[] array, int arrayIndex);
    bool IsReadOnly { get; }
```

```
    void Add(T item);
    bool Remove (T item);
    void Clear();
}
```

Необобщенный интерфейс `ICollection` похож в том, что предоставляет коллекцию с возможностью подсчета, но не предлагает функциональности для изменения списка или проверки членства элементов:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo (Array array, int index);
}
```

В необобщенном интерфейсе также определены свойства для содействия в синхронизации (см. главу 14) — они были помещены в обобщенную версию, потому что безопасность в отношении потоков больше не считается внутренне присущей коллекции.

Оба интерфейса довольно прямолинейны в реализации. Если реализуется интерфейс `ICollection<T>`, допускающий только чтение, то методы `Add`, `Remove` и `Clear` должны генерировать исключение `NotSupportedException`.

Эти интерфейсы обычно реализуются в сочетании с интерфейсом `IList` или `IDictionary`.

Интерфейсы `IList<T>` и `IList`

`IList<T>` — стандартный интерфейс для коллекций, поддерживающих индексацию по позиции. В дополнение к функциональности, унаследованной от интерфейсов `ICollection<T>` и `IEnumerable<T>`, он предлагает возможность чтения/записи элемента по позиции (через индексатор) и вставки/удаления по позиции:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}
```

Методы `IndexOf` выполняют линейный поиск в списке, возвращая `-1`, если указанный элемент не найден.

Необобщенная версия `IList` имеет большее количество членов, поскольку она меньше наследует из `ICollection`:

```
public interface IList : ICollection, IEnumerable
{
    object this [int index] { get; set; }
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
```

```
int Add      (object value);
void Clear();
bool Contains (object value);
int IndexOf  (object value);
void Insert   (int index, object value);
void Remove   (object value);
void RemoveAt (int index);
}
```

Метод `Add` необобщенного интерфейса `IList` возвращает целочисленное значение — индекс вновь добавленного элемента. Напротив, метод `Add` интерфейса `ICollection<T>` имеет возвращаемый тип `void`.

Универсальный класс `List<T>` является наиболее типичной реализацией обоих интерфейсов `IList<T>` и `IList`. Массивы в C# также реализуют обобщенную и необобщенную версии `IList` (хотя методы добавления или удаления элементов скрыты через явную реализацию интерфейса и в случае вызова генерируют исключение `NotSupportedException`).



При попытке доступа в многомерный массив через индексатор `IList` генерируется исключение `ArgumentException`. Такая опасность возникает при написании методов вроде показанного ниже:

```
public object FirstOrDefault (IList list)
{
    if (list == null || list.Count == 0) return null;
    return list[0];
}
```

Код может выглядеть “пуленепробиваемым”, однако он приведет к генерации исключения в случае вызова с многомерным массивом. Проверить во время выполнения, является ли массив многомерным, можно с помощью следующего кода (за дополнительными сведениями обращайтесь в главу 19):

```
list.GetType().IsArray && list.GetType().GetArrayRank() > 1
```

Интерфейсы `IReadOnlyCollection<T>` и `IReadOnlyList<T>`

В .NET определены интерфейсы коллекций и списков, открывающие доступ лишь к членам, которые требуются для операций, допускающих только чтение.

```
public interface IReadOnlyCollection<out T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
}

public interface IReadOnlyList<out T> : IReadOnlyCollection<T>,
                                         IEnumerable<T>, IEnumerable
{
    T this[int index] { get; }
}
```

Из-за того, что параметр типа таких интерфейсов используется только в выходных позициях, он помечается как *ковариантный*. В итоге появляется возможность трактовать список кошек, например, как список животных, предназначенный только для чтения. Напротив, в `IList<T>` тип `T` не помечается как ковариантный, потому что он присутствует и во входных, и в выходных позициях.



Описанные интерфейсы являются *представлением* коллекции или списка, предназначенным только для чтения; лежащая в основе реализация по-прежнему может допускать запись. Большинство записываемых (*изменяемых*) коллекций реализуют как интерфейсы только для чтения, так и интерфейсы для чтения/записи.

Помимо возможности работы с коллекциями ковариантным образом интерфейсы, допускающие только чтение, позволяют классу открывать доступ к представлению только для чтения закрытой записываемой коллекции. Мы продемонстрируем это вместе с лучшим решением в разделе “Класс `ReadOnlyCollection<T>`” далее в главе.

Интерфейс `IReadOnlyList<T>` отображается на тип Windows Runtime по имени `IVectorView<T>`.

Класс `Array`

Класс `Array` является неявным базовым классом для всех одномерных и многомерных массивов, а также одним из наиболее фундаментальных типов, реализующих стандартные интерфейсы коллекций. Класс `Array` обеспечивает унификацию типов, так что общий набор методов доступен всем массивам независимо от их объявления или типа элементов.

По причине настолько фундаментального характера массивов язык C# предлагает явный синтаксис для их объявления и инициализации, который был описан в главах 2 и 3. Когда массив объявляется с применением синтаксиса C#, среда CLR неявно создает подтип класса `Array` за счет синтеза *псевдотипа*, подходящего для размерностей и типов элементов массива. Псевдотип реализует типизированные необобщенные интерфейсы коллекций вроде `IList<string>`.

Среда CLR трактует типы массивов специальным образом также при конструировании, выделяя им непрерывный участок в памяти. Это делает индексацию в массивах высокоэффективной, но препятствует изменению их размеров в будущем.

Класс `Array` реализует интерфейсы коллекций вплоть до `IList<T>` в обеих формах — обобщенной и необобщенной. Однако сам интерфейс `IList<T>` реализован явно, чтобы сохранить открытый интерфейс `Array` свободным от методов, подобных `Add` или `Remove`, которые генерируют исключение на коллекциях фиксированной длины, таких как массивы. Класс `Array` в действительности предлагает статический метод `Resize`, хотя он работает путем создания нового массива и копирования в него всех элементов. Вдобавок к такой неэффективности ссылки на массив в других местах программы будут по-прежнему указывать на его исходную версию. Более удачное решение для коллекций с из-

меняемыми размерами предусматривает использование класса `List<T>` (описанного в следующем разделе).

Массив может содержать элементы, имеющие тип значения или ссылочный тип. Элементы типа значения хранятся в массиве на месте, а потому массив из трех элементов типа `long` (по 8 байтов каждое) будет занимать непрерывный участок памяти размером 24 байта. С другой стороны, элементы ссылочного типа занимают только объем, требующийся для ссылки (4 байта в 32-разрядной среде или 8 байтов в 64-разрядной среде). На рис. 7.2 показано распределение в памяти для приведенного ниже кода:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

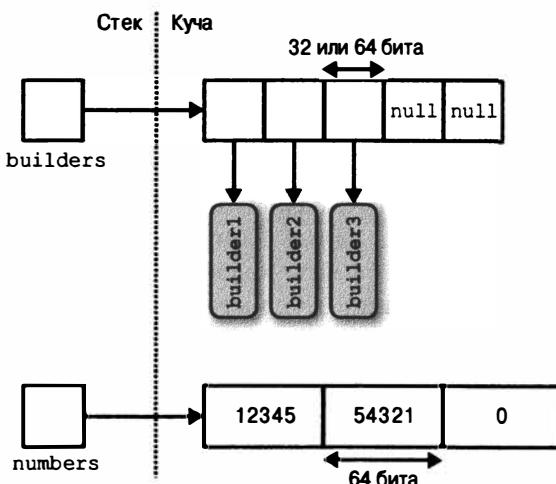


Рис. 7.2. Массивы в памяти

Из-за того, что `Array` представляет собой класс, массивы (сами по себе) всегда являются ссылочными типами независимо от типа элементов массива. Это значит, что оператор `arrayB = arrayA` даст в результате две переменные, которые ссылаются на один и тот же массив. Аналогично два разных массива всегда будут приводить к отрицательному результату при проверке эквивалентности — если только вы не задействуете *компаратор структурной эквивалентности*, который сравнивает каждый элемент массива:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };
Console.WriteLine (a1 == a2);           // False
Console.WriteLine (a1.Equals (a2));     // False
IStructuralEquatable sel = a1;
Console.WriteLine (sel.Equals (a2,
    StructuralComparisons.StructuralEqualityComparer)); // True
```

Дубликат массива можно создавать вызовом метода `Clone`: `arrayB = arrayA.Clone()`. Тем не менее, результатом будет поверхностная (неглубокая) копия, означающая копирование только участка памяти, в котором представлен собственно массив. Если массив содержит объекты типов значений, тогда копируются сами значения; если же массив содержит объекты ссылочных типов, то копируются только ссылки (в итоге давая два массива с членами, которые ссылаются на одни и те же объекты). На рис. 7.3 показан эффект от добавления в пример следующего кода:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

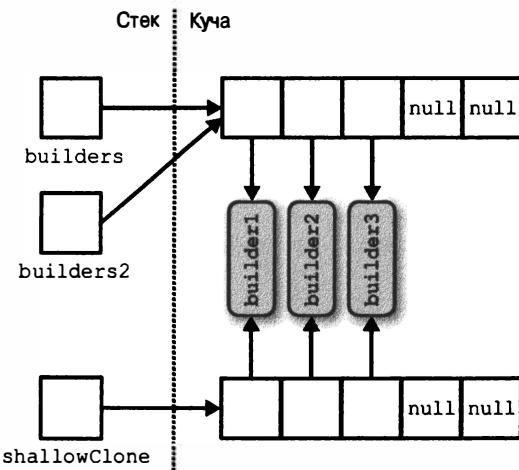


Рис. 7.3. Поверхностное копирование массива

Чтобы создать глубокую копию, при которой объекты ссылочных типов дублируются, потребуется пройти в цикле по массиву и клонировать каждый его элемент вручную. Те же самые правила применимы к другим типам коллекций .NET.

Хотя класс `Array` предназначен в первую очередь для использования с 32-битными индексаторами, он также располагает ограниченной поддержкой 64-битных индексаторов (позволяя массиву теоретически адресовать до 264 элементов) через несколько методов, которые принимают параметры типов `Int32` и `Int64`. На практике эти перегруженные версии бесполезны, т.к. CLR не разрешает ни одному объекту — включая массивы — занимать более 2 Гбайт (как в 32-разрядной, так и в 64-разрядной среде).



Многие методы в классе `Array`, которые вы, возможно, ожидали видеть как методы экземпляра, на самом деле являются статическими методами. Такое довольно странное проектное решение означает, что при поиске подходящего метода в `Array` следует просматривать и статические методы, и методы экземпляра.

Конструирование и индексация

Простейший способ создания и индексации массивов предусматривает применение языковых конструкций C#:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

В качестве альтернативы можно создать экземпляр массива динамически, вызвав метод `Array.CreateInstance`, что позволяет указывать тип элементов и ранг (количество измерений) во время выполнения — равно как и создавать массивы с индексацией, начинающейся не с нуля, задавая нижнюю границу. Массивы с индексацией, начинающейся не с нуля, не совместимы с поддерживаемой в .NET общеязыковой спецификацией (Common Language Specification — CLS) и не должны быть доступными как открытые члены в библиотеке, которая может потребляться программой, написанной на языке F# или Visual Basic.

Методы `GetValue` и `SetValue` позволяют получать доступ к элементам в динамически созданном массиве (они также работают с обычными массивами):

```
// Создать строковый массив длиной 2 элемента:
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                                // → a[0] = "hi";
a.SetValue ("there", 1);                            // → a[1] = "there";
string s = (string) a.GetValue (0);                  // → s = a[0];

// Можно также выполнить приведение к массиву C#:
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Созданные динамическим образом массивы с индексацией, начинающейся с нуля, могут быть приведены к массиву C# совпадающего или совместимого типа (совместимого согласно стандартным правилам вариантиности массивов). Например, если `Apple` является подклассом `Fruit`, то массив `Apple[]` может быть приведен к `Fruit[]`. В результате возникает вопрос о том, почему в качестве унифицированного типа массива вместо класса `Array` не был выбран `object[]`? Дело в том, что массив `object[]` несовместим с многомерными массивами и массивами типов значений (и массивами с индексацией, начинающейся не с нуля). Массив `int[]` не может быть приведен к `object[]`. Таким образом, для полной унификации типов требуется класс `Array`.

Методы `GetValue` и `SetValue` также работают с массивами, созданными компилятором, и они полезны при написании методов, которые имеют дело с массивом любого типа и ранга. Для многомерных массивов они принимают массив индексаторов:

```
public object GetValue (params int[] indices)
public void SetValue (object value, params int[] indices)
```

Следующий метод выводит на экран первый элемент любого массива независимо от ранга (количества измерений):

```

void WriteFirstValue (Array a)
{
    Console.Write (a.Rank + "-dimensional; "); // размерность массива
    // Массив индексаторов будет автоматически инициализироваться всеми нулями,
    // поэтому его передача в метод GetValue или SetValue приведет к получению
    // или установке основанного на нуле (т.е. первого) элемента в массиве.

    int[] indexers = new int[a.Rank];
    Console.WriteLine ("First value is " + a.GetValue (indexers));
}
void Demo()
{
    int[] oneD = { 1, 2, 3 };
    int[,] twoD = { {5,6}, {8,9} };
    WriteFirstValue (oneD); // одномерный; первое значение равно 1
    WriteFirstValue (twoD); // двумерный; первое значение равно 5
}

```



Для работы с массивами неизвестного типа, но с известным рангом, обобщения предлагают более простое и эффективное решение:

```

void WriteFirstValue<T> (T[] array)
{
    Console.WriteLine (array[0]);
}

```

Метод SetValue генерирует исключение, если элемент имеет тип, несовместимый с массивом. Когда экземпляр массива создан, либо посредством языкового синтаксиса, либо с помощью Array.CreateInstance, его элементы автоматически инициализируются своими стандартными значениями. Для массивов с элементами ссылочных типов это означает занесение в них null, а для массивов с элементами типов значений — побитовое “обнуление” членов. Класс Array предоставляет такую функциональность и по запросу через метод Clear:

```
public static void Clear (Array array, int index, int length);
```

Данный метод не влияет на размер массива, что отличается от обычного использования метода Clear (такого как в ICollection<T>.Clear), при котором коллекция сокращается до нуля элементов.

Перечисление

С массивами легко выполнять перечисление с помощью оператора foreach:

```

int[] myArray = { 1, 2, 3 };
foreach (int val in myArray)
    Console.WriteLine (val);

```

Перечисление можно также проводить с применением метода Array.ForEach, определенного следующим образом:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Здесь используется делегат Action с приведенной ниже сигнатурой:

```
public delegate void Action<T> (T obj);
```

А вот как выглядит первый пример, переписанный с применением метода `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

Начиная с версии C# 12, код можно еще больше упростить с помощью *выражения коллекции*:

```
Array.ForEach ([ 1, 2, 3 ], Console.WriteLine);
```

Длина и ранг

Класс `Array` предоставляет следующие методы и свойства для запрашивания длины и ранга:

```
public int GetLength (int dimension);
public long GetLongLength (int dimension);

public int Length { get; }
public long LongLength { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; } // Возвращает количество измерений в массиве (ранг)
```

Методы `GetLength` и `GetLongLength` возвращают длину для заданного измерения (0 для одномерного массива), а свойства `Length` и `LongLength` — количество элементов в массиве (включая все измерения).

Методы `GetLowerBound` и `GetUpperBound` удобны при работе с массивами, индексы которых начинаются не с нуля. Метод `GetUpperBound` возвращает сумму значений, возвращенных методами `GetLowerBound` и `GetLength` для любого заданного измерения.

Поиск

Класс `Array` предлагает набор методов для нахождения элементов внутри одномерного массива.

Методы `BinarySearch`

Для быстрого поиска заданного элемента в отсортированном массиве.

Методы `IndexOf/LastIndex`

Для поиска заданного элемента в несортированном массиве.

Методы `Find/FindLast/FindIndex/FindLastIndex/FindAll/Exists/TrueForAll`

Для поиска элемента или элементов, удовлетворяющих заданному предикату `Predicate<T>`, в несортированном массиве.

Ни один из методов поиска в массиве не генерирует исключение в случае, если указанное значение не найдено. Взамен, когда элемент не найден, методы, возвращающие целочисленное значение, возвращают -1 (предполагая, что индекс массива начинается с нуля), а методы, возвращающие обобщенный тип, возвращают стандартное значение для этого типа (скажем, 0 для `int` или `null` для `string`).

Методы двоичного поиска характеризуются высокой скоростью, но работают только с отсортированными массивами и требуют, чтобы элементы сравнивались на предмет *порядка*, а не просто *эквивалентности*. С такой целью методы двоичного поиска могут принимать объект, реализующий интерфейс `IComparer` или `IComparer<T>`, который позволяет принимать решения по упорядочению (см. раздел “Подключение протоколов эквивалентности и порядка” далее в главе). Он должен быть согласован с любым компаратором, используемым при исходной сортировке массива. Если компаратор не предоставляется, то будет применен стандартный алгоритм упорядочения типа, основанный на его реализации `IComparable/IComparable<T>`.

Методы `IndexOf` и `LastIndexOf` выполняют простое перечисление по массиву, возвращая первый (или последний) элемент, который совпадает с заданным значением.

Методы поиска на основе предикатов позволяют управлять *совпадением* заданного элемента с помощью метода делегата или лямбда-выражения. Предикат представляет собой просто делегат, принимающий объект и возвращающий `true` или `false`:

```
public delegate bool Predicate<T> (T object);
```

В следующем примере выполняется поиск в массиве строк имени, содержащего букву “a”:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, ContainsA);
Console.WriteLine (match);                                // Jack
ContainsA (string name) { return name.Contains ("a"); }
```

Ниже показан тот же пример, сокращенный с помощью лямбда-выражения:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a")); // Jack
```

Метод `FindAll` возвращает массив всех элементов, удовлетворяющих предикату. На самом деле он эквивалентен методу `Enumerable.Where` из пространства имен `System.Linq` за исключением того, что `FindAll` возвращает массив совпадающих элементов, а не перечисление `IEnumerable<T>` таких элементов.

Метод `Exists` возвращает `true`, если член массива удовлетворяет заданному предикату, и он эквивалентен методу `Any` класса `System.Linq.Enumerable`.

Метод `TrueForAll` возвращает `true`, если все элементы удовлетворяют заданному предикату, и он эквивалентен методу `All` класса `System.Linq.Enumerable`.

Сортировка

Класс `Array` имеет следующие встроенные методы сортировки:

```
// Для сортировки одиночного массива:
public static void Sort<T> (T[] array);
public static void Sort      (Array array);
// Для сортировки пары массивов:
```

```
public static void Sort<TKey, TValue> (TKey[] keys, TValue[] items);  
public static void Sort (Array keys, Array items);
```

Каждый из указанных методов дополнительно перегружен, чтобы принимать также описанные далее аргументы:

```
int index // Начальный индекс, с которого должна стартовать сортировка  
int length // Количество элементов, подлежащих сортировке  
IComparer<T> comparer // Объект, принимающий решения по упорядочению  
Comparison<T> comparison // Делегат, принимающий решения по упорядочению
```

Ниже демонстрируется простейший случай использования метода Sort:

```
int[] numbers = { 3, 2, 1 };  
Array.Sort (numbers); // Массив теперь содержит { 1, 2, 3 }
```

Методы, принимающие пару массивов, работают путем переупорядочения элементов каждого массива в tandemе, основываясь на решениях по упорядочению из первого массива. В следующем примере числа и соответствующие им слова сортируются в числовом порядке:

```
int[] numbers = { 3, 2, 1 };  
string[] words = { "three", "two", "one" };  
Array.Sort (numbers, words);  
  
// Массив numbers теперь содержит { 1, 2, 3 }  
// Массив words теперь содержит { "one", "two", "three" }
```

Метод Array.Sort требует, чтобы элементы в массиве реализовывали интерфейс IComparable (см. раздел “Сравнение порядка” в главе 6). Это означает возможность сортировки для большинства встроенных типов C# (таких как целочисленные типы из предыдущего примера). Если элементы по своей сути несравнимы или нужно переопределить стандартное упорядочение, то методу Sort понадобится предоставить собственный поставщик сравнения, который будет сообщать относительные позиции двух элементов. Решить задачу можно двумя способами:

- посредством вспомогательного объекта, который реализует интерфейс IComparer/IComparer<T> (см. раздел “Подключение протоколов эквивалентности и порядка” далее в главе);

- посредством делегата Comparison:

```
public delegate int Comparison<T> (T x, T y);
```

Делегат Comparison следует той же семантике, что и метод IComparer<T>.CompareTo: если x находится перед y, то возвращается отрицательное целое число; если x располагается после y, тогда возвращается положительное целое число; если x и y имеют одну и ту же позицию сортировки, то возвращается 0.

В следующем примере мы сортируем массив целых чисел так, чтобы нечетные числа шли первыми:

```
int[] numbers = { 1, 2, 3, 4, 5 };  
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);  
  
// Массив numbers теперь содержит { 1, 3, 5, 2, 4 }
```



В качестве альтернативы вызову метода Sort можно применять операции OrderBy и ThenBy языка LINQ. В отличие от Array.Sort операции LINQ не изменяют исходный массив, а взамен выдают отсортированный результат в новой последовательности IEnumerable<T>.

Обращение порядка следования элементов

Приведенные ниже методы класса Array изменяют порядок следования всех или части элементов массива на противоположный:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

Копирование

Класс Array предлагает четыре метода для выполнения поверхностного копирования: Clone, CopyTo, Copy и ConstrainedCopy. Первые два являются методами экземпляра, а последние два — статическими методами.

Метод Clone возвращает полностью новый (поверхностно скопированный) массив. Методы CopyTo и Copy копируют непрерывное подмножество элементов массива. Копирование многомерного прямоугольного массива требует отображения многомерного индекса на линейный индекс. Например, позиция [1, 1] в массиве 3×3 представляется индексом 4 на основе следующего вычисления: $1 \times 3 + 1$. Исходный и целевой диапазоны могут перекрываться без возникновения проблем.

Метод ConstrainedCopy выполняет *атомарную* операцию: если все запрошенные элементы не могут быть успешно скопированы (например, из-за ошибки, связанной с типом), то производится откат всей операции.

Класс Array также предоставляет метод AsReadOnly, который возвращает оболочку, предотвращающую переустановку элементов.

Преобразование и изменение размера

Метод Array.ConvertAll создает и возвращает новый массив элементов типа TOutput, вызывая во время копирования элементов предоставленный делегат Converter. Делегат Converter определен следующим образом:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

Приведенный ниже код преобразует массив чисел с плавающей точкой в массив целых чисел:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));
// Массив wholes содержит { 1, 2, 2 }
```

Метод Resize создает новый массив и копирует в него элементы, возвращая новый массив через ссылочный параметр. Любые ссылки на исходный массив в других объектах остаются неизмененными.



Пространство имен `System.Linq` предлагает дополнительный набор расширяющих методов, которые подходят для преобразования массивов. Такие методы возвращают экземпляр реализации интерфейса `IEnumerable<T>`, который можно преобразовать обратно в массив с помощью метода `ToArray` класса `Enumerable`.

Списки, очереди, стеки и наборы

.NET предоставляет базовый набор конкретных классов коллекций, которые реализуют интерфейсы, описанные в настоящей главе. В этом разделе внимание сосредоточено на списковых коллекциях (в противоположность словарным коллекциям, обсуждаемым в разделе “Словари” далее в главе). Как и в случае с ранее рассмотренными интерфейсами, обычно доступен выбор между обобщенной и необобщенной версиями каждого типа. В плане гибкости и производительности обобщенные классы имеют преимущество, делая свои необобщенные аналоги избыточными и предназначеными только для обеспечения обратной совместимости. Ситуация с интерфейсами коллекций иная: их необобщенные версии все еще иногда полезны.

Из описанных в данном разделе классов наиболее часто используется обобщенный класс `List`.

Классы `List<T>` и `ArrayList`

Обобщенный класс `List` и необобщенный класс `ArrayList` представляют массив объектов с возможностью динамического изменения размера и относятся к числу самых часто применяемых классов коллекций. Класс `ArrayList` реализует интерфейс `IList`, в то время как класс `List<T>` — интерфейсы `IList` и `IList<T>` (а также `IReadOnlyList<T>` — новую версию, допускающую только чтение). В отличие от массивов все интерфейсы реализованы открытым образом, а методы вроде `Add` и `Remove` открыты и работают совершенно ожидаемо.

Классы `List<T>` и `ArrayList` функционируют за счет поддержки внутреннего массива объектов, который заменяется более крупным массивом при достижении предельной емкости. Добавление элементов производится эффективно (потому что в конце обычно есть свободные позиции), но вставка элементов может оказаться медленной (т.к. все элементы после точки вставки должны быть сдвинуты для освобождения позиции), как и удаление элементов (особенно поблизости к началу). Аналогично массивам поиск будет эффективным, если метод `BinarySearch` используется со списком, который был отсортирован, но в противном случае он не эффективен, поскольку каждый элемент должен проверяться индивидуально.



Класс `List<T>` работает в несколько раз быстрее класса `ArrayList`, если `T` является типом значения, потому что `List<T>` избегает накладных расходов, связанных с упаковкой и распаковкой элементов.

Классы `List<T>` и `ArrayList` предоставляют конструкторы, которые принимают существующую коллекцию элементов — они копируют каждый элемент из нее в новый экземпляр `List<T>` или `ArrayList`:

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
    public List();
    public List(IEnumerable<T> collection);
    public List(int capacity);

    // Добавление и вставка:
    public void Add(T item);
    public void AddRange(IEnumerable<T> collection);
    public void Insert(int index, T item);
    public void InsertRange(int index, IEnumerable<T> collection);

    // Удаление:
    public bool Remove(T item);
    public void RemoveAt(int index);
    public void RemoveRange(int index, int count);
    public int RemoveAll(Predicate<T> match);

    // Индексация:
    public T this[int index] { get; set; }
    public List<T> GetRange(int index, int count);
    public Enumerator<T> GetEnumerator();

    // Экспортирование, копирование и преобразование:
    public T[] ToArray();
    public void CopyTo(T[] array);
    public void CopyTo(T[] array, int arrayIndex);
    public void CopyTo(int index, T[] array, int arrayIndex, int count);
    public ReadOnlyCollection<T> AsReadOnly();
    public List<TOutput> ConvertAll<TOutput>(Converter<T, TOutput>
                                                converter);

    // Другие:
    public void Reverse(); // Меняет порядок следования элементов
                          // в списке на противоположный
    public int Capacity { get; set; } // Инициализирует расширение внутреннего массива
    public void TrimExcess(); // Усекает внутренний массив до
                           // фактического количества элементов
    public void Clear(); // Удаляет все элементы, так что Count=0
}

public delegate TOutput Converter<TInput, TOutput>(TInput input);
```

В дополнение к указанным членам класс `List<T>` предлагает версии экземпляра для всех методов поиска и сортировки из класса `Array`.

В показанном ниже коде демонстрируется работа свойств и методов `List` (примеры поиска и сортировки приводились в разделе “Класс `Array`” ранее в главе):

```
List<string> words = new List<string>(); // Новый список типа string
words.Add("melon");
words.Add("avocado");
words.AddRange(["banana", "plum"]);
words.Insert(0, "lemon"); // Вставить в начало
```

```
words.InsertRange (0, ["peach", "nashi"]); // Вставить в начало
words.Remove ("melon");
words.RemoveAt (3); // Удалить 4-й элемент
words.RemoveRange (0, 2); // Удалить первые 2 элемента

// Удалить все строки, начинающиеся с n:
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]); // первое слово
Console.WriteLine (words [words.Count - 1]); // последнее слово
foreach (string s in words) Console.WriteLine (s); // все слова
List<string> subset = words.GetRange (1, 2); // слова со 2-го по 3-е
string[] wordsArray = words.ToArray(); // Создает новый типизированный массив

// Копировать первые два элемента в конец существующего массива:
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCaseWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

Необобщенный класс `ArrayList` требует довольно неуклюжих приведений:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Такие приведения не могут быть проверены компилятором; скажем, представленный далее код скомпилируется успешно, но потерпит неудачу во время выполнения:

```
int first = (int) al [0]; // Исключение во время выполнения
```



Класс `ArrayList` функционально похож на класс `List<object>`. Оба класса удобны, когда необходим список элементов смешанных типов, которые не разделяют общий базовый тип (кроме `object`). В таком случае выбор `ArrayList` может обеспечить преимущество, если в отношении списка должна использоваться рефлексия (см. главу 18). Рефлексию реализовать проще с помощью необобщенного класса `ArrayList`, чем посредством `List<object>`.

Импортировав пространство имен `System.Linq`, экземпляр `ArrayList` можно преобразовать в обобщенный `List` за счет вызова метода `Cast` и затем `ToList`:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

`Cast` и `ToList` — расширяющие методы в классе `System.Linq.Enumerable`.

Класс `LinkedList<T>`

Класс `LinkedList<T>` представляет обобщенный двусвязный список (рис. 7.4). Двусвязный список — это цепочка узлов, в которой каждый узел ссылается на предыдущий узел, следующий узел и действительный элемент. Его главное преимущество заключается в том, что элемент может быть эффективно вставлен в любое место списка, т.к. подобное действие требует только создания нового узла и обновления нескольких ссылок. Однако поиск позиции вставки может оказаться медленным ввиду отсутствия в связном списке встроенного механизма для прямой индексации; должен производиться обход каждого узла, и двоичный поиск невозможен.

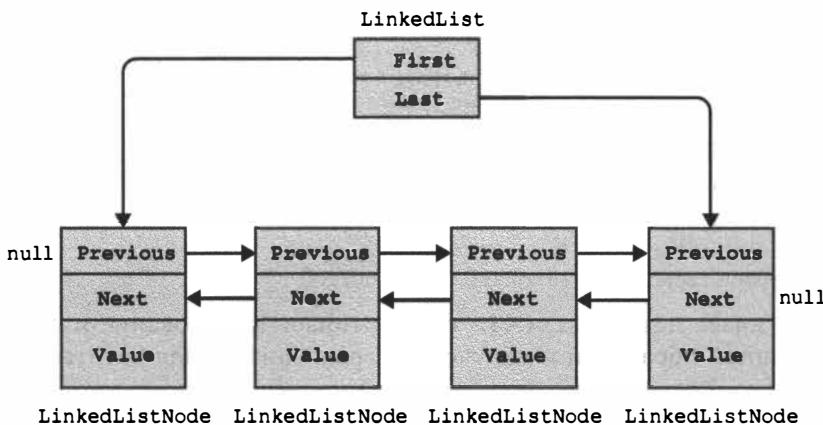


Рис. 7.4. Класс `LinkedList<T>`

Класс `LinkedList<T>` реализует интерфейсы `IEnumerable<T>` и `ICollection<T>` (а также их необобщенные версии), но не `IList<T>`, поскольку доступ по индексу не поддерживается. Узлы списка реализованы с помощью такого класса:

```
public sealed class LinkedListNode<T>
{
    public LinkedList<T> List { get; }
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```

При добавлении узла можно указывать его позицию либо относительно другого узла, либо относительно начала/конца списка. Класс `LinkedList<T>` предоставляет для этого следующие методы:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);
```

```
public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Для удаления элементов предлагаются похожие методы:

```
public void Clear();
public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

Класс `LinkedList<T>` имеет внутренние поля для отслеживания количества элементов в списке, а также для представления головы и хвоста списка. Доступ к таким полям обеспечивается с помощью следующих открытых свойств:

```
public int Count { get; }           // Быстрое
public LinkedListNode<T> First { get; }    // Быстрое
public LinkedListNode<T> Last { get; }     // Быстрое
```

Класс `LinkedList<T>` также поддерживает показанные далее методы поиска (каждый из них требует, чтобы список был внутренне перечислимым):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Наконец, класс `LinkedList<T>` поддерживает копирование в массив для индексированной обработки и получение перечислителя для работы оператора `foreach`:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Ниже демонстрируется применение класса `LinkedList<string>`:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                      // do
tune.AddLast ("so");                       // do - so
tune.AddAfter (tune.First, "re");           // do - re - so
tune.AddAfter (tune.First.Next, "mi");       // do - re - mi - so
tune.AddBefore (tune.Last, "fa");            // do - re - mi - fa - so
tune.RemoveFirst();                         // re - mi - fa - so
tune.RemoveLast();                          // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                      // re - fa
tune.AddFirst (miNode);                    // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);
```

Классы `Queue<T>` и `Queue`

Классы `Queue<T>` и `Queue` — это структуры данных FIFO (first-in first-out — первым пришел, первым обслужен; т.е. очередь), предоставляющие методы `Enqueue` (добавление элемента в конец очереди) и `Dequeue` (извлечение и удаление элемента с начала очереди). Также имеется метод `Peek`, предназначенный

для возвращения элемента с начала очереди без его удаления, и свойство Count (удобно для проверки, существуют ли элементы в очереди, перед их извлечением).

Хотя очереди являются перечислимыми, они не реализуют интерфейс IList<T>/IList, потому что доступ к членам напрямую по индексу никогда не производится. Тем не менее, есть метод ToArray, который предназначен для копирования элементов в массив, где к ним возможен произвольный доступ:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Queue();
    public Queue (IEnumerable<T> collection); //Копирует существующие элементы
    public Queue (int capacity);           // Сокращает количество автома-
                                              // тических изменений размера
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public T Dequeue();
    public void Enqueue (T item);
    public Enumerator<T> GetEnumerator(); // Для поддержки оператора foreach
    public T Peek();
    public T[] ToArray();
    public void TrimExcess();
}
```

Вот пример использования класса Queue<int>:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray();           // Экспортирует в массив
Console.WriteLine (q.Count);        // "2"
Console.WriteLine (q.Peek());        // "10"
Console.WriteLine (q.Dequeue());     // "10"
Console.WriteLine (q.Dequeue());     // "20"
Console.WriteLine (q.Dequeue());     // Генерируется исключение (очередь пуста)
```

Внутренне очереди реализованы с применением массива, который при необходимости расширяется, что очень похоже на обобщенный класс List. Очередь поддерживает индексы, которые указывают непосредственно на начальный и хвостовой элементы; таким образом, помещение и извлечение из очереди являются очень быстрыми операциями (кроме случая, когда требуется внутреннее изменение размера).

Классы Stack<T> и Stack

Классы Stack<T> и Stack — это структуры данных LIFO (last-in first-out — последним пришел, первым обслужен; т.е. стек), которые предоставляют методы Push (добавление элемента на верхушку стека) и Pop (извлечение и удаление элемента из верхушки стека). Также определены недеструктивный метод Peek, свойство Count и метод ToArray для экспорта данных в массив с целью произвольного доступа к ним:

```

public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Stack();
    public Stack (IEnumerable<T> collection); // Копирует существующие элементы
    public Stack (int capacity); // Сокращает количество автоматических изменений размера
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public Enumerator<T> GetEnumerator(); // Для поддержки оператора foreach
    public T Peek();
    public T Pop();
    public void Push (T item);
    public T[] ToArray();
    public void TrimExcess();
}

```

В следующем примере демонстрируется использование класса Stack<int>:

```

var s = new Stack<int>();
s.Push (1); // Содержимое стека: 1
s.Push (2); // Содержимое стека: 1,2
s.Push (3); // Содержимое стека: 1,2,3
Console.WriteLine (s.Count); // Выводит 3
Console.WriteLine (s.Peek()); // Выводит 3, содержимое стека: 1,2,3
Console.WriteLine (s.Pop()); // Выводит 3, содержимое стека: 1,2
Console.WriteLine (s.Pop()); // Выводит 2, содержимое стека: 1
Console.WriteLine (s.Pop()); // Выводит 1, содержимое стека: <пусто>
Console.WriteLine (s.Pop()); // Генерируется исключение (стек пуст)

```

Внутренне стеки реализованы с помощью массива, который при необходимости расширяется, что очень похоже на Queue<T> и List<T>.

Класс BitArray

Класс BitArray представляет собой коллекцию сжатых значений bool с динамически изменяющимся размером. С точки зрения затрат памяти класс BitArray более эффективен, чем простой массив bool и обобщенный список элементов bool, поскольку каждое значение занимает только один бит; в противном случае тип bool требует по одному байту на значение.

Индексатор BitArray читает и записывает индивидуальные биты:

```

var bits = new BitArray(2);
bits[1] = true;

```

Доступны четыре метода для выполнения побитовых операций (And, Or, Xor и Not). Все они кроме последнего принимают еще один экземпляр BitArray:

```

bits.Xor (bits); // Побитовое исключающее ИЛИ экземпляра bits с самим собой
Console.WriteLine (bits[1]); // False

```

Классы HashSet<T> и SortedSet<T>

Классы HashSet<T> и SortedSet<T> обладают следующим отличительными особенностями:

- их методы Contains выполняются быстро, применяя поиск на основе хеширования;
- они не хранят дублированные элементы и молча игнорируют запросы на добавление дубликатов;
- доступ к элементам по позициям невозможен.

Коллекция SortedSet<T> хранит элементы в упорядоченном виде, тогда как HashSet<T> — нет.

Общность типов HashSet<T> и SortedSet<T> обеспечивается интерфейсом ISet<T>. Начиная с .NET 5, эти классы также реализуют интерфейс по имени IReadOnlySet<T>, который реализуется и типами неизменяемых множеств (см. раздел “Неизменяемые коллекции” далее в главе).

Коллекция HashSet<T> реализована с помощью хеш-таблицы, в которой хранятся только ключи, а SortedSet<T> — посредством красно-черного дерева.

Обе коллекции реализуют интерфейс ICollection<T> и предлагают вполне ожидаемые методы, такие как Contains, Add и Remove. Вдобавок предусмотрен метод удаления на основе предиката по имени RemoveWhere.

В следующем коде из существующей коллекции конструируется экземпляр HashSet<char>, затем выполняется проверка членства и перечисление коллекции (обратите внимание на отсутствие дубликатов):

```
var letters = new HashSet<char> ("the quick brown fox");
Console.WriteLine (letters.Contains ('t'));           // True
Console.WriteLine (letters.Contains ('j'));           // False
foreach (char c in letters) Console.Write (c);       // the quickbrownfx
```

(Передача значения string конструктору HashSet<char> объясняется тем, что тип string реализует интерфейс IEnumerable<char>.)

Самый большой интерес вызывают операции над множествами. Приведенные ниже методы операций над множествами являются *деструктивными*, т.к. они модифицируют набор:

```
public void UnionWith      (IEnumerable<T> other);    // Добавляет
public void IntersectWith   (IEnumerable<T> other);    // Удаляет
public void ExceptWith     (IEnumerable<T> other);    // Удаляет
public void SymmetricExceptWith (IEnumerable<T> other); // Удаляет
```

тогда как следующие методы просто запрашивают набор и потому деструктивными не считаются:

```
public bool IsSubsetOf      (IEnumerable<T> other);
public bool IsProperSubsetOf (IEnumerable<T> other);
public bool IsSupersetOf    (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps        (IEnumerable<T> other);
public bool SetEquals       (IEnumerable<T> other);
```

Метод `UnionWith` добавляет все элементы из второго набора в первоначальный набор (исключая дубликаты). Метод `IntersectWith` удаляет элементы, которые не находятся сразу в обоих наборах. Вот как можно извлечь все гласные из набора символов:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);           // euio
```

Метод `ExceptWith` удаляет указанные элементы из исходного набора. Ниже показано, каким образом удалить все гласные из набора:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);           // th qckbrwnfx
```

Метод `SymmetricExceptWith` удаляет все элементы кроме тех, которые являются уникальными в одном или в другом наборе:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);           // quicklazy
```

Обратите внимание, что поскольку типы `HashSet<T>` и `SortedSet<T>` реализуют интерфейс `IEnumerable<T>`, в качестве аргумента в любом методе операции над множествами можно использовать другой тип набора (или коллекции).

Тип `SortedSet<T>` предлагает все члены типа `HashSet<T>` и вдобавок следующие члены:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

Кроме того, конструктор класса `SortedSet<T>` принимает дополнительный параметр типа `IComparer<T>` (отличающийся от *компаратора эквивалентности*).

Ниже приведен пример загрузки в `SortedSet<char>` тех же букв, что и ранее:

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);           // bcefhi knoqrtuwx
```

Исходя из этого, получить буквы между “f” и “i” в наборе можно так:

```
foreach (char c in letters.GetViewBetween ('f', 'i'))
Console.Write (c);                                // fhi
```

Словари

Словарь — это коллекция, в которой каждый элемент является парой “ключ/значение”. Словари чаще всего применяются для поиска и представления сортirованных списков.

В .NET определен стандартный протокол для словарей через интерфейсы `IDictionary` и `IDictionary<TKey, TValue>`, а также набор универсальных классов словарей. Упомянутые классы различаются в следующих отношениях:

- хранятся ли элементы в отсортированной последовательности;
- можно ли получать доступ к элементам по позиции (индексу) и по ключу;
- является тип обобщенным или необобщенным;
- является тип быстрым или медленным при извлечении элементов по ключу из крупного словаря.

В табл. 7.1 представлена сводка по всем классам словарей, а также описаны отличия в перечисленных выше аспектах. Значения времени указаны в миллисекундах и основаны на выполнении 50000 операций в словаре с целочисленными ключами и значениями на персональном компьютере с процессором 1,5 ГГц. (Разница в производительности между обобщенными и необобщенными версиями для одной и той же структуры коллекции объясняется упаковкой и связана только с элементами типов значений.)

С помощью записи “большое О” время извлечения можно описать следующим образом:

- $O(1)$ для `Hashtable`, `Dictionary` и `OrderedDictionary`;
- $O(\log n)$ для `SortedDictionary` и `SortedList`;
- $O(n)$ для `ListDictionary` (и несловарных типов, таких как `List<T>`).

где n — количество элементов в коллекции.

Интерфейс `IDictionary<TKey, TValue>`

Интерфейс `IDictionary<TKey, TValue>` определяет стандартный протокол для всех коллекций, основанных на парах “ключ/значение”. Он расширяет интерфейс `ICollection<T>`, добавляя методы и свойства для доступа к элементам на основе ключей произвольных типов:

```
public interface IDictionary <TKey, TValue> :  
    ICollection <KeyValuePair <TKey, TValue>>, IEnumerable  
{  
    bool ContainsKey (TKey key);  
    bool TryGetValue (TKey key, out TValue value);  
    void Add (TKey key, TValue value);  
    bool Remove (TKey key);  
  
    TValue this [TKey key] { get; set; } // Основной индексатор - по ключу  
    ICollection <TKey> Keys { get; } // Возвращает только ключи  
    ICollection <TValue> Values { get; } // Возвращает только значения  
}
```



Существует также интерфейс по имени `IReadOnlyDictionary<TKey, TValue>`, в котором определено подмножество членов словаря, допускающих только чтение.

Таблица 7.1. Классы словарей

Тип	Внутренняя структура	Поддерживается ли извлечение по индексу?	Накладные расходы, связанные с памятью (среднее количество байтов на элемент)	Скорость: произвольная вставка	Скорость: последовательная вставка	Скорость: извлечение по ключу
Несортированные						
Dictionary<K, V>	Хеш-таблица	Нет	22	30	30	20
Hashtable	Хеш-таблица	Нет	38	50	50	30
ListDictionary	Связный список	Нет	36	50 000	50 000	50 000
OrderedDictionary	Хеш-таблица плюс массив	Да	59	70	70	40
Сортированные						
SortedDictionary<K, V>	Красно-черное дерево	Нет	20	130	100	120
SortedList<K, V>	Пара массивов	Да	2	3 300	30	40
SortedList	Пара массивов	Да	27	4 500	100	180

Чтобы добавить элемент в словарь, необходимо либо вызвать метод Add, либо воспользоваться средством доступа set индекса — в последнем случае элемент добавляется в словарь, если такой ключ в словаре отсутствует (или производится обновление элемента, если ключ присутствует). Дублированные ключи запрещены во всех реализациях словарей, поэтому вызов метода Add два раза с тем же самым ключом приводит к генерации исключения.

Для извлечения элемента из словаря применяется либо индексатор, либо метод TryGetValue. Если ключ не существует, тогда индексатор генерирует исключение, в то время как метод TryGetValue возвращает false. Можно явно проверить членство, вызвав метод ContainsKey; однако за это придется заплатить двумя поисками, если элемент впоследствии будет извлекаться.

Перечисление прямо по `IDictionary< TKey, TValue >` возвращает последовательность структур `KeyValuePair`:

```
public struct KeyValuePair < TKey, TValue >
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```

С помощью свойств `Keys/Values` словаря можно организовать перечисление только по ключам или только по значениям.

Мы продемонстрируем использование интерфейса `IDictionary< TKey, TValue >` с обобщенным классом `Dictionary` в следующем разделе.

Интерфейс `IDictionary`

Необобщенный интерфейс `IDictionary` в принципе является таким же, как интерфейс `IDictionary< TKey, TValue >`, за исключением двух важных функциональных отличий. Эти отличия необходимо понимать, потому что `IDictionary` присутствует в унаследованном коде (а местами и в самой библиотеке .NET BCL):

- извлечение несуществующего ключа через индексатор дает в результате null (не приводя к генерации исключения);
- членство проверяется с помощью метода `Contains`, но не `ContainsKey`.

Перечисление по необобщенному интерфейсу `IDictionary` возвращает последовательность структур `DictionaryEntry`:

```
public struct DictionaryEntry
{
    public object Key { get; set; }
    public object Value { get; set; }
}
```

Классы `Dictionary< TKey, TValue >` и `Hashtable`

Обобщенный класс `Dictionary` представляет одну из наиболее часто применяемых коллекций (наряду с коллекцией `List< T >`). Для хранения ключей и значений класс `Dictionary` использует структуру данных в форме хеш-таблицы, а также характеризуется высокой скоростью работы и эффективностью.



Необобщенная версия `Dictionary< TKey, TValue >` называется `Hashtable`; необобщенного класса, который бы имел имя “`Dictionary`”, не существует. Когда мы ссылаемся просто на `Dictionary`, то имеем в виду обобщенный класс `Dictionary< TKey, TValue >`.

Класс `Dictionary` реализует обобщенный и необобщенный интерфейсы `IDictionary`, причем обобщенный интерфейс `IDictionary` открыт. Фактически `Dictionary` является “учебной” реализацией обобщенного интерфейса `IDictionary`.

Ниже показано, как с ним работать:

```
var d = new Dictionary<string, int>();

d.Add("One", 1);
d["Two"] = 2;    // Добавляет в словарь, потому что "two" пока отсутствует
d["Two"] = 22;   // Обновляет словарь, т.к. "two" уже присутствует
d["Three"] = 3;

Console.WriteLine (d["Two"]);           // Выводит "22"
Console.WriteLine (d.ContainsKey ("One")); // True (быстрая операция)
Console.WriteLine (d.ContainsKeyValue (3)); // True (медленная операция)
int val = 0;
if (!d.TryGetValue ("onE", out val))
    Console.WriteLine ("No val"); // "No val" (чувствительно к регистру)

// Три разных способа перечисления словаря:
foreach (KeyValuePair<string, int> kv in d)      // One; 1
    Console.WriteLine (kv.Key + "; " + kv.Value); // Two; 22
                                                // Three; 3

foreach (string s in d.Keys) Console.Write (s); // OneTwoThree
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i); // 1223
```

Лежащая в основе `Dictionary` хеш-таблица преобразует ключ каждого элемента в целочисленный хеш-код — псевдоуникальное значение — и затем применяет алгоритм для преобразования хеш-кода в хеш-ключ. Такой хеш-ключ используется внутренне для определения, к какому “сегменту” относится запись. Если сегмент содержит более одного значения, тогда в нем производится линейный поиск. Хорошая хеш-функция не стремится возвращать строго уникальные хеш-коды (что обычно невозможно); она старается вернуть хеш-коды, которые равномерно распределены в пространстве 32-битных целых чисел. Это позволяет избежать сценария с получением нескольких очень крупных (и неэффективных) сегментов.

Благодаря своей возможности определения эквивалентности ключей и получения хеш-кодов словарь может работать с ключами любого типа. По умолчанию эквивалентность определяется с помощью метода `object.Equals` ключа, а псевдоуникальный хеш-код получается через метод `GetHashCode` ключа. Такое поведение можно изменить, либо переопределив указанные методы, либо предоставив при конструировании словаря объект, который реализует интерфейс `IEqualityComparer`.

Распространенный случай применения предусматривает указание нечувствительного к регистру компаратора эквивалентности, когда используются строковые ключи:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Мы обсудим данный момент более подробно в разделе “Подключение протоколов эквивалентности и порядка” далее в главе.

Как и со многими другими типами коллекций, производительность словаря можно несколько улучшить за счет указания в конструкторе ожидаемого размера коллекции, избегая или снижая тем самым потребность во внутренних операциях изменения размера.

Необобщенная версия имеет имя `Hashtable` и функционально подобна, не считая отличий, которые являются результатом открытия ею необобщенного интерфейса `IDictionary`, как обсуждалось ранее.

Недостаток `Dictionary` и `Hashtable` связан с тем, что элементы не отсортированы. Кроме того, первоначальный порядок, в котором добавлялись элементы, не предохраняется. Как и со всеми словарями, дублированные ключи не разрешены.



Когда в 2005 году появились обобщенные коллекции, команда разработчиков CLR решила именовать их согласно тому, что они представляют (`Dictionary`, `List`), а не тому, как они реализованы внутренне (`Hashtable`, `ArrayList`). Хотя такой подход обеспечивает свободу изменения реализации в будущем, он также означает, что в имени больше не отражается *контракт производительности* (зачастую являющийся самым важным критерием при выборе одного вида коллекции из нескольких доступных).

Класс `OrderedDictionary`

Класс `OrderedDictionary` — необобщенный словарь, который хранит элементы в порядке их добавления. С помощью `OrderedDictionary` получать доступ к элементам можно и по индексам, и по ключам.



Класс `OrderedDictionary` не является *отсортированным* словарем.

Класс `OrderedDictionary` представляет собой комбинацию классов `Hashtable` и `ArrayList`. Таким образом, он обладает всей функциональностью `Hashtable`, а также имеет функции вроде `RemoveAt` и целочисленный индексатор. Кроме того, класс `OrderedDictionary` открывает доступ к свойствам `Keys` и `Values`, которые возвращают элементы в их исходном порядке.

Класс `OrderedDictionary` появился в .NET 2.0 и, как ни странно, его обобщенной версии не предусмотрено.

Классы `ListDictionary` и `HybridDictionary`

Для хранения лежащих в основе данных в классе `ListDictionary` применяется односвязный список. Он не обеспечивает сортировку, хотя предохраняет исходный порядок элементов. С большими списками класс `ListDictionary` работает исключительно медленно. Единственным “предметом гордости” можно считать его эффективность в случае очень маленьких списков (до 10 элементов).

Класс `HybridDictionary` — это `ListDictionary`, который автоматически преобразуется в `Hashtable` при достижении определенного размера, решая проблемы низкой производительности класса `ListDictionary`. Идея заключается в том, чтобы обеспечить невысокое потребление памяти для мелких словарей и хорошую производительность для крупных словарей. Однако, учитывая накладные расходы, которые сопровождают переход от одного класса к другому, а также тот факт, что класс `Dictionary` довольно неплох в любом сценарии, вполне разумно использовать `Dictionary` с самого начала.

Оба класса доступны только в необобщенной форме.

Отсортированные словари

Библиотека .NET BCL предлагает два класса словарей, которые внутренне устроены так, что их содержимое всегда сортируется по ключу:

- `SortedDictionary<TKey, TValue>`
- `SortedList<TKey, TValue>1`

(В настоящем разделе мы будем сокращать `<TKey, TValue>` до `<, >`.)

Класс `SortedDictionary<, >` применяет красно-черное дерево — структуру данных, которая спроектирована так, что работает одинаково хорошо в любом сценарии вставки либо извлечения.

Класс `SortedList<, >` внутренне реализован с помощью пары упорядоченных массивов, обеспечивая высокую производительность извлечения (посредством двоичного поиска), но низкую производительность вставки (поскольку существующие значения должны сдвигаться, чтобы освободить место под новый элемент).

Класс `SortedDictionary<, >` намного быстрее класса `SortedList<, >` при вставке элементов в произвольном порядке (особенно в случае крупных списков). Тем не менее, класс `SortedList<, >` обладает дополнительной возможностью: доступом к элементам по индексу и по ключу. Благодаря отсортированному списку можно переходить непосредственно к *n*-ному элементу в отсортированной последовательности (с помощью индексатора на свойствах `Keys`/`Values`). Чтобы сделать то же самое с помощью `SortedDictionary<, >`, потребуется вручную пройти через *n* элементов. (В качестве альтернативы можно было бы написать класс, комбинирующий отсортированный словарь и список.)

Ни одна из трех коллекций не допускает наличие дублированных ключей (как и все словари).

¹ Существует также функционально идентичная ему необобщенная версия по имени `SortedList`.

В следующем примере используется рефлексия с целью загрузки всех методов, определенных в классе `System.Object`, внутрь отсортированного списка с ключами по именам, после чего производится перечисление их ключей и значений:

```
// Класс MethodInfo находится в пространстве имен System.Reflection
var sorted = new SortedList <string, MethodInfo>();
foreach (MethodInfo m in typeof (object).GetMethods())
    sorted [m.Name] = m;
foreach (string name in sorted.Keys)
    Console.WriteLine (name);
foreach (MethodInfo m in sorted.Values)
    Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

Ниже показаны результаты первого перечисления:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

А вот результаты второго перечисления:

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

Обратите внимание, что словарь наполняется посредством своего индексатора. Если заменить метод `Add`, то сгенерируется исключение, т.к. в классе `object`, на котором осуществляется рефлексия, метод `Equals` перегружен, и добавить в словарь тот же самый ключ два раза не удастся. В случае использования индексатора элемента, добавляемый позже, переписывает элемент, добавленный раньше, предотвращая возникновение такой ошибки.



Можно сохранять несколько членов одного ключа, делая каждый элемент значения списком:

```
SortedList <string, List<MethodInfo>>
```

Расширяя рассматриваемый пример, следующий код извлекает объект `MethodInfo` с ключом "GetHashCode", точно как в случае обычного словаря:

```
Console.WriteLine (sorted ["GetHashCode"]); // Int32 GetHashCode()
```

Весь написанный до сих пор код также будет работать с классом `SortedDictionary<,>`. Однако показанные ниже две строки кода, которые извлекают последний ключ и значение, работают только с отсортированным списком:

```
Console.WriteLine (sorted.Keys [sorted.Count - 1]); // ToString
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // True
```

Настраиваемые коллекции и посредники

Классы коллекций, которые обсуждались в предшествующих разделах, удобны своей возможностью непосредственного создания экземпляров, но они не позволяют управлять тем, что происходит, когда элемент добавляется или удаляется из коллекции. При работе со строго типизированными коллекциями в приложении периодически возникает необходимость в таком контроле, например:

- для запуска события, когда элемент добавляется или удаляется;
- для обновления свойств из-за добавления или удаления элемента;
- для обнаружения “несанкционированной” операции добавления/удаления и генерации исключения (скажем, если операция нарушает какое-то бизнес-правило).

Именно по указанным причинам библиотека .NET BCL предоставляет классы коллекций, определенные в пространстве имен `System.Collections.ObjectModel`. В сущности, они являются посредниками, или оболочками, реализующими интерфейс `IList<T>` либо `IDictionary<,>`, которые переадресуют вызовы методам внутренней коллекции. Каждая операция `Add`, `Remove` или `Clear` проходит через виртуальный метод, действующий в качестве “шлюза”, когда он переопределен.

Классы настраиваемых коллекций обычно применяются для коллекций, открытых публично; например, в классе `System.Windows.Form` официально открыта коллекция элементов управления.

Классы `Collection<T>` и `CollectionBase`

Класс `Collection<T>` является настраиваемой оболочкой для `List<T>`. Помимо реализации интерфейсов `IList<T>` и `IList` в нем определены четыре дополнительных виртуальных метода и защищенное свойство:

```
public class Collection<T> :  
    IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable  
{  
    // ...  
  
    protected virtual void ClearItems();  
    protected virtual void InsertItem (int index, T item);  
    protected virtual void RemoveItem (int index);  
    protected virtual void SetItem (int index, T item);  
  
    protected IList<T> Items { get; }  
}
```

Виртуальные методы предоставляют шлюз, с помощью которого можно “привязаться” с целью изменения или расширения нормального поведения списка. Защищенное свойство `Items` позволяет реализующему коду получать прямой доступ во “внутренний список” — это используется для внесения изменений внутренне без запуска виртуальных методов.

Виртуальные методы переопределять не обязательно; их можно оставить незатронутыми, если только не возникает потребность в изменении стандартного

поведения списка. В следующем примере показан типичный “скелет” программы, в которой применяется класс Collection<T>:

```
Zoo zoo = new Zoo();
zoo.Animals.Add (new Animal ("Kangaroo", 10));
zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);

public class Animal
{
    public string Name;
    public int Popularity;

    public Animal (string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : Collection <Animal>
{
    // AnimalCollection - уже полностью функционирующий список животных.
    // Никакого дополнительного кода не требуется.
}

public class Zoo      // Класс, который откроет доступ к AnimalCollection.
{
    // Обычно он может иметь дополнительные члены.
    public readonly AnimalCollection Animals = new AnimalCollection();
}
```

Как здесь видно, класс AnimalCollection не обладает большей функциональностью, чем простой класс List<Animal>; его роль заключается в том, чтобы предоставить базу для будущего расширения. В целях иллюстрации мы добавим к Animal свойство Zoo, так что экземпляр животного может ссылаться на зоопарк (zoo), где оно содержится, и переопределим все виртуальные методы в Collection<Animal> для автоматической поддержки данного свойства:

```
public class Animal
{
    public string Name;
    public int Popularity;
    public Zoo Zoo { get; internal set; }
    public Animal(string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : Collection <Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
    protected override void InsertItem (int index, Animal item)
    {
        base.InsertItem (index, item);
        item.Zoo = zoo;
    }
}
```

```

protected override void SetItem (int index, Animal item)
{
    base.SetItem (index, item);
    item.Zoo = zoo;
}
protected override void RemoveItem (int index)
{
    this [index].Zoo = null;
    base.RemoveItem (index);
}
protected override void ClearItems()
{
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
}
}
public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}

```

Класс `Collection<T>` также имеет конструктор, который принимает существующую реализацию `IList<T>`. В отличие от других классов коллекций передаваемый список не копируется, а для него создается посредник, т.е. последующие изменения будут отражаться в оболочке `Collection<T>` (хотя и без запуска виртуальных методов `Collection<T>`). И наоборот, изменения, внесенные через `Collection<T>`, будут воздействовать на лежащий в основе список.

Класс CollectionBase

Класс `CollectionBase` — это необобщенная версия `Collection<T>`. Он поддерживает большинство тех же возможностей, что и `Collection<T>`, но менее удобен в использовании.

Вместо шаблонных методов `InsertItem`, `RemoveItem`, `SetItem` и `ClearItem` класс `CollectionBase` имеет методы “привязки”, что удваивает количество требуемых методов: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear` и `OnClearComplete`. Поскольку класс `CollectionBase` не является обобщенным, при создании его подклассов понадобится также реализовать типизированные методы — как минимум, типизированный индексатор и метод `Add`.

Классы KeyedCollection<TKey, TItem> и DictionaryBase

Класс `KeyedCollection<TKey, TItem>` представляет собой подкласс класса `Collection<TItem>`. Определенная функциональность в него добавлена, а определенная — удалена. К добавленной функциональности относится возможность доступа к элементам по ключу, почти как в словаре. Удаление функциональности касается устраниния возможности создавать посредника для собственного внутреннего списка.

Коллекция с ключами имеет некоторое сходство с классом `OrderedDictionary` в том, что комбинирует линейный список с хеш-таблицей. Тем не менее, в отличие от `OrderedDictionary` коллекция с ключами не реализует интерфейс `IDictionary` и не поддерживает концепцию пары “ключ/значение”. Взамен ключи получаются из самих элементов: через абстрактный метод `GetKeyForItem`. Это означает, что перечисление по коллекции с ключами производится точно так же, как в обычном списке.

Лучше всего воспринимать `KeyedCollection< TKey, TItem >` как класс `Collection< TItem >` с добавочным быстрым поиском по ключу.

Поскольку коллекция с ключами является подклассом класса `Collection<>`, она наследует всю функциональность `Collection<>` кроме возможности указания существующего списка при конструировании. В классе `KeyedCollection< TKey, TItem >` определены дополнительные члены, как показано ниже:

```
public abstract class KeyedCollection < TKey, TItem > : Collection < TItem >
{
    // ...

    protected abstract TKey GetKeyForItem(TItem item);
    protected void ChangeItemKey(TItem item, TKey newKey);

    // Быстрый поиск по ключу - является дополнением к поиску по индексу
    public TItem this[TKey key] { get; }

    protected IDictionary< TKey, TItem > Dictionary { get; }
}
```

Метод `GetKeyForItem` переопределяется для получения ключа элемента из лежащего в основе объекта. Метод `ChangeItemKey` должен вызываться, если свойство ключа элемента изменяется, чтобы обновить внутренний словарь. Свойство `Dictionary` возвращает внутренний словарь, применяемый для реализации поиска, который создается при добавлении первого элемента. Такое поведение можно изменить, указав в конструкторе порог создания, что отсрочит создание внутреннего словаря до момента, когда будет достигнуто пороговое значение (а тем временем при поступлении запроса элемента по ключу будет выполняться линейный поиск). Веская причина, по которой порог создания не указывается, объясняется тем, что наличие допустимого словаря может быть полезно в получении коллекции `ICollection<>` ключей через свойство `Keys` класса `Dictionary`. Затем данная коллекция может быть передана открытому свойству.

Самое распространенное использование класса `KeyedCollection<>` связано с предоставлением коллекции элементов, доступных как по индексу, так и по имени. В целях демонстрации мы реализуем класс `AnimalCollection` в виде `KeyedCollection< string, Animal >`:

```
public class Animal
{
    string name;
    public string Name
    {
        get { return name; }
```

```

        set {
            if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
            name = value;
        }
    }
    public int Popularity;
    public Zoo Zoo { get; internal set; }
    public Animal (string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}
public class AnimalCollection : KeyedCollection <string, Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
    internal void NotifyNameChange (Animal a, string newName) =>
        this.ChangeItemKey (a, newName);
    protected override string GetKeyForItem (Animal item) => item.Name;
    //Следующие методы должны быть реализованы так же, как в предыдущем примере
    protected override void InsertItem (int index, Animal item)... 
    protected override void SetItem (int index, Animal item)... 
    protected override void RemoveItem (int index)... 
    protected override void ClearItems()...
}

public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}

```

Вот как можно задействовать класс AnimalCollection:

```

Zoo zoo = new Zoo();
zoo.Animals.Add (new Animal ("Kangaroo", 10));
zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
Console.WriteLine (zoo.Animals [0].Popularity);           // 10
Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity); // 20
zoo.Animals ["Kangaroo"].Name = "Mr Roo";
Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);     // 10

```

Класс DictionaryBase

Необобщенная версия коллекции KeyedCollection называется DictionaryBase. Этот унаследованный класс существенно отличается принятым в нем подходом: он реализует интерфейс IDictionary и подобно классу CollectionBase применяет множество неуклюжих методов привязки: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear и OnClearComplete (и дополнительно OnGet). Главное преимущество реализации IDictionary вместо принятия подхода с KeyedCollection состоит в том, что для получения ключей не требуется создавать его подкласс. Но поскольку основным назначением DictionaryBase

является создание подклассов, в итоге какие-либо преимущества вообще отсутствуют. Улучшенная модель в `KeyedCollection` почти наверняка объясняется тем фактом, что данный класс был написан несколькими годами позже, с оглядкой на прошлый опыт. Класс `DictionaryBase` лучше всего рассматривать как предназначенный для обратной совместимости.

Класс `ReadOnlyCollection<T>`

Класс `ReadOnlyCollection<T>` — это оболочка, или *посредник*, который является представлением коллекции, доступным только для чтения. Он полезен в ситуации, когда нужно разрешить классу открывать доступ только для чтения к коллекции, которую данный класс может внутренне обновлять.

Конструктор класса `ReadOnlyCollection<T>` принимает входную коллекцию, на которую будет поддерживаться постоянная ссылка. Он не создает статическую копию входной коллекции, а потому последующие изменения входной коллекции будут видны через оболочку, допускающую только чтение.

В целях иллюстрации предположим, что классу необходимо предоставить открытый доступ только для чтения к списку строк по имени `Names`. Вот как мы могли бы поступить:

```
public class Test
{
    List<string> names = new List<string>();
    public IReadonlyList<string> Names => names;
}
```

Хотя `Names` возвращает интерфейс только для чтения, потребитель по-прежнему может во время выполнения предпринять приведение вниз к `List<string>` или `IList<string>` и затем вызывать `Add`, `Remove` или `Clear` на списке. Класс `ReadOnlyCollection<T>` обеспечивает более надежное решение:

```
public class Test
{
    List<string> names = new List<string>();
    public ReadOnlyCollection<string> Names { get; private set; }
    public Test() => Names = new ReadOnlyCollection<string> (names);
    public void AddInternally() => names.Add ("test");
}
```

Теперь изменять список имен могут только члены класса `Test`:

```
Test t = new Test();
Console.WriteLine (t.Names.Count);           // 0
t.AddInternally();
Console.WriteLine (t.Names.Count);           // 1
t.Names.Add ("test");                      // Ошибка на этапе компиляции
((IList<string>) t.Names).Add ("test");    // Генерируется исключение
                                            // NotSupportedException
```

Неизменяемые коллекции

Только что было описано, каким образом `ReadOnlyCollection<T>` создает представление коллекции, допускающее только чтение. Ограничение возможности записывания в коллекцию (ее изменения) или в любой другой объект упрощает программное обеспечение и уменьшает количество ошибок.

Неизменяемые коллекции расширяют данный принцип, предлагая коллекции, которые после инициализации вообще не могут быть модифицированы. Если вам нужно добавить элемент в неизменяемую коллекцию, тогда вы должны создать новый экземпляр коллекции, оставив старый экземпляр незатронутым.

Неизменяемость является отличительной чертой функционального программирования и обеспечивает следующие преимущества:

- она устраниет крупный класс ошибок, ассоциированных с изменением состояния;
- она значительно упрощает реализацию параллелизма и многопоточной обработки, избегая большинства проблем с безопасностью в отношении потоков, которые мы рассмотрим в главах 14, 21 и 22;
- она облегчает понимание кода.

Недостаток неизменяемости в том, что когда необходимо внести изменение, вам придется создавать целиком новый объект. В итоге снижается производительность, хотя в текущем разделе мы обсудим стратегии смягчения последствий, включая возможность повторного использования частей первоначальной структуры.

Неизменяемые коллекции являются частью .NET (в .NET Framework они доступны через NuGet-пакет `System.Collections.Immutable`). Все коллекции определены в пространстве имен `System.Collections.Immutable`:

Тип	Внутренняя структура
<code>ImmutableArray<T></code>	Массив
<code>ImmutableList<T></code>	AVL-дерево
<code>ImmutableDictionary<K, V></code>	AVL-дерево
<code>ImmutableHashSet<T></code>	AVL-дерево
<code>ImmutableSortedDictionary<K, V></code>	AVL-дерево
<code>ImmutableSortedSet<T></code>	AVL-дерево
<code>ImmutableStack<T></code>	Связный список
<code>ImmutableQueue<T></code>	Связный список

Типы `ImmutableArray<T>` и `ImmutableList<T>` представляют собой неизменяемые версии `List<T>`. Оба они делают ту же самую работу, но с разными характеристиками производительности, которые мы обсудим в разделе “Неизменяемые коллекции и производительность” далее в главе.

Неизменяемые коллекции предоставляют доступ к открытому интерфейсу подобно своим изменяемым аналогам. Ключевое отличие в том, что методы, которые выглядят как изменяющие коллекцию (вроде Add или Remove), не модифицируют первоначальную коллекцию, а взамен возвращают новую коллекцию с добавленным или удаленным элементом. Это называется *неразрушающим изменением*.



Неизменяемые коллекции предотвращают добавление и удаление элементов; они не препятствуют изменению *самых* элементов. Чтобы получить все преимущества неизменяемости, вы должны обеспечить наличие в неизменяемой коллекции только неизменяемых элементов.

Создание неизменяемых коллекций

Каждый тип неизменяемой коллекции предлагает метод Create<T>(), который принимает необязательные начальные значения и возвращает инициализированную неизменяемую коллекцию:

```
ImmutableArray<int> array = ImmutableArray.Create<int> (1, 2, 3);
```

Кроме того, каждая коллекция предлагает метод CreateRange<T>, выполняющий такую же работу, как и Create<T>; разница в том, что его параметром типа является I Enumerable<T>, а не params T[].

Создавать неизменяемую коллекцию можно и из существующей реализации I Enumerable<T>, используя подходящие расширяющие методы (ToImmutableArray, ToImmutableList, ToImmutableDictionary и т.д.):

```
var list = new[] { 1, 2, 3 }.ToImmutableList();
```

Манипулирование неизменяемыми коллекциями

Метод Add возвращает новую коллекцию, содержащую существующие элементы плюс новый элемент:

```
var oldList = ImmutableList.Create<int> (1, 2, 3);
ImmutableList<int> newList = oldList.Add (4);
Console.WriteLine (oldList.Count);           // 3 (не изменилось)
Console.WriteLine (newList.Count);           // 4
```

Метод Remove работает в той же манере, возвращая новую коллекцию, из которой удален элемент.

Многократное добавление или удаление элементов подобным образом неэффективно, потому что для каждой операции добавления или удаления создается новая неизменяемая коллекция. Более удачное решение предусматривает вызов метода AddRange (или RemoveRange), принимающего реализацию I Enumerable<T> с элементами, которые будут добавлены или удалены за один присест:

```
var anotherList = oldList.AddRange ([4, 5, 6]);
```

В неизменяемом списке и массиве также определены методы `Insert` и `InsertRange` для вставки элементов по специальному индексу, метод `RemoveAt` для удаления элемента по индексу и метод `RemoveAll`, который удаляет на основе предиката.

Построители

Для более сложных потребностей в инициализации каждый класс неизменяемой коллекции определяет аналог *построителя*. Построители представляют собой классы, которые функционально эквивалентны изменяемой коллекции и обладают сходными характеристиками производительности. После того, как данные инициализированы, вызов `.ToImmutable()` на построителе возвращает неизменяемую коллекцию:

```
ImmutableArray<int>.Builder builder = ImmutableArray.CreateBuilder<int>();
builder.Add (1);
builder.Add (2);
builder.Add (3);
builder.RemoveAt (0);
ImmutableArray<int> myImmutable = builder.ToImmutable();
```

Построители можно также применять для *пакетирования* множества обновлений, подлежащих внесению в существующую неизменяемую коллекцию:

```
var builder2 = myImmutable.ToBuilder();
builder2.Add (4);           // Эффективная операция
builder2.Remove (2);        // Эффективная операция
...
// Дополнительные изменения в построителе...
// Возвратить новую неизменяемую коллекцию, в которой применены все изменения:
ImmutableArray<int> myImmutable2 = builder2.ToImmutable();
```

Неизменяемые коллекции и производительность

Большинство неизменяемых коллекций внутренне используют *AVL-дерево* (сбалансированное двоичное дерево поиска; аббревиатура AVL образована по первым буквам фамилий его создателей Г.М. Адельсона-Вельского и Е.М. Ландиса — *прим. пер.*), которое дает возможность операциям добавления/удаления повторно задействовать части первоначальной внутренней структуры, не создавая коллекцию заново. В итоге накладные расходы операций добавления/удаления снижаются с потенциально *огромных* (в случае крупных коллекций) до *умеренно больших*, но за счет замедления операций чтения. В результате большинство неизменяемых коллекций оказываются медленнее своих изменяемых аналогов при чтении и записи.

Наиболее серьезно страдает тип `ImmutableList<T>`, операции чтения и добавления в котором от 10 до 200 раз медленнее, чем в `List<T>` (в зависимости от размера списка). Вот почему существует тип `ImmutableArray<T>`: за счет применения внутри себя массива он избегает накладных расходов, связанных с операциями чтения (для которых он сопоставим по производительности с обычным изменяемым массивом). Обратной стороной является тот факт, что операции добавления в нем гораздо медленнее, чем (даже) в `ImmutableList<T>`, потому что ничего из первоначальной структуры не может использоваться повторно.

Следовательно, тип `ImmutableArray<T>` желательно применять, когда вам нужна высокая производительность **чтения**, и вы не ожидаете множества обращений к `Add` или `Remove` (без использования построителя):

Тип	Производительность чтения	Производительность добавления
<code>ImmutableList<T></code>	Низкая	Низкая
<code>ImmutableArray<T></code>	Очень высокая	Очень низкая



Вызов метода `Remove` на экземпляре `ImmutableArray` сопряжен с более высокими затратами, чем вызов `Remove` на экземпляре `List<T>` (даже в худшем случае удаления первого элемента), поскольку выделение памяти под новую коллекцию создает дополнительную нагрузку на сборщик мусора.

Хотя неизменяемые коллекции в целом влекут за собой потенциально значительное снижение производительности, важно удерживать общую величину в перспективе. На обычном портативном компьютере операция добавления для экземпляра `ImmutableList` с миллионом элементов по-прежнему может длиться менее микросекунды, а операция чтения — менее 100 нс. И если вам необходимо выполнять операции записи в цикле, то вы можете избежать накопления накладных расходов за счет применения построителя.

Уменьшить накладные расходы помогают также следующие факторы.

- Неизменяемость позволяет облегчить распараллеливание (см. главу 22), так что вы можете задействовать все доступные ядра. Распараллеливание с изменяемым состоянием легко приводит к ошибкам и требует использования блокировок или параллельных коллекций, что наносит ущерб производительности.
- Благодаря неизменяемости вам не придется создавать “защитные копии” коллекций или структур данных, чтобы препятствовать непредвиденным изменениям. Это было фактором в пользу применения неизменяемых коллекций при написании недавних порций Visual Studio.
- В большинстве типичных программ некоторые коллекции содержать достаточно большое количество элементов, чтобы разница имела значение.

Помимо Visual Studio с использованием неизменяемых коллекций был построен хорошо работающий набор инструментальных средств Microsoft Roslyn, демонстрируя тем самым, что преимущества могут перевешивать затраты.

Замороженные коллекции

Начиная с .NET 8, пространство имен `System.Collections.Frozen` содержит следующие два класса коллекций, допускающие только чтение:

- `FrozenDictionary<TKey, TValue>`
- `FrozenSet<T>`

Они похожи на классы `ImmutableDictionary<K, V>` и `ImmutableHashSet<T>`, но в них отсутствуют методы неразрушающего изменения (вроде `Add` или `Remove`), что обеспечивает высокую оптимизацию производительности чтения. Для создания замороженной коллекции необходимо сначала создать другую коллекцию или последовательность, а затем вызвать расширяющий метод `ToFrozenDictionary` или `ToFrozenSet`:

```
int[] numbers = { 10, 20, 30 };
FrozenSet<int> frozen = numbers.ToFrozenSet();
Console.WriteLine(frozen.Contains(10)); // True
```

Замороженные коллекции великолепно подходят для поиска, который инициализируется в начале программы и затем используется на протяжении всей жизни приложения:

```
class Disassembler
{
    public readonly static IReadOnlyDictionary<string, string> OpCodeLookup =
        new Dictionary<string, string>()
    {
        { "ADC", "Add with Carry" },
        { "ADD", "Add" },
        { "AND", "Logical AND" },
        { "ANDN", "Logical AND NOT" },
        ...
    }.ToFrozenDictionary();
    ...
}
```

Замороженные коллекции реализуют стандартные интерфейсы словаря/набора, включая их версии, доступные только для чтения. В этом примере мы представили `FrozenDictionary<string, string>` как поле типа `IReadOnlyDictionary<string, string>`.

Подключение протоколов эквивалентности и порядка

В разделах “Сравнение эквивалентности” и “Сравнение порядка” главы 6 были описаны стандартные протоколы .NET, которые привносят в тип возможность эквивалентности, хеширования и сравнения. Тип, который реализует упомянутые протоколы, может корректно функционировать в словаре или отсортированном списке. В частности:

- тип, для которого методы `Equals` и `GetHashCode` возвращают осмысленные результаты, может использоваться в качестве ключа в `Dictionary` или `Hashtable`;
- тип, который реализует `IComparable/IComparable<T>`, может применяться в качестве ключа в любом *отсортированном* словаре или списке.

Стандартная реализация эквивалентности или сравнения типа обычно отражает то, что является наиболее “естественным” для данного типа. Тем не менее,

иногда стандартное поведение не подходит. Может понадобиться словарь, в котором ключ типа `string` трактуется без учета регистра. Или же может потребоваться список заказчиков, отсортированный по их почтовым индексам. По этой причине в .NET также определен соответствующий набор “подключаемых” протоколов. Подключаемые протоколы служат двум целям:

- они позволяют переключаться на альтернативное поведение эквивалентности или сравнения;
- они позволяют использовать словарь или отсортированную коллекцию с типом ключа, который не обладает внутренней возможностью эквивалентности или сравнения.

Подключаемые протоколы состоят из указанных ниже интерфейсов.

- `IEqualityComparer` и `IEqualityComparer<T>`
 - выполняют подключаемое *сравнение эквивалентности и хеширование*;
 - распознаются классами `Hashtable` и `Dictionary`.
- `IComparer` и `IComparer<T>`
 - выполняют подключаемое *сравнение порядка*;
 - распознаются отсортированными словарями и коллекциями, а также методом `Array.Sort`.

Каждый интерфейс доступен в обобщенной и необобщенной формах. Интерфейсы `IEqualityComparer` также имеют стандартную реализацию в классе по имени `EqualityComparer`.

Кроме того, существуют интерфейсы `IStructuralEquatable` и `IStructuralComparable`, которые позволяют выполнять структурные сравнения для классов и массивов.

Интерфейсы `IEqualityComparer` и `EqualityComparer`

Компаратор эквивалентности позволяет переключаться на нестандартное поведение эквивалентности и хеширования главным образом для классов `Dictionary` и `Hashtable`.

Вспомним требования к словарю, основанному на хеш-таблице. Для любого заданного ключа он должен отвечать на два следующих вопроса.

- Является ли указанный ключ таким же, как другой ключ?
- Какой целочисленный хеш-код имеет указанный ключ?

Компаратор эквивалентности отвечает на такие вопросы путем реализации интерфейсов `IEqualityComparer`:

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y);
    int GetHashCode (T obj);
}
```

```
public interface IEqualityComparer // Необобщенная версия
{
    bool Equals (object x, object y);
    int GetHashCode (object obj);
}
```

Для написания специального компаратора необходимо реализовать один или оба интерфейса (реализация обоих интерфейсов обеспечивает максимальную степень взаимодействия). Поскольку это несколько утомительно, в качестве альтернативы можно создать подкласс абстрактного класса EqualityComparer, определение которого показано ниже:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
    IEqualityComparer<T>
{
    public abstract bool Equals (T x, T y);
    public abstract int GetHashCode (T obj);

    bool IEqualityComparer.Equals (object x, object y);
    int IEqualityComparer.GetHashCode (object obj);

    public static EqualityComparer<T> Default { get; }
}
```

Класс EqualityComparer реализует оба интерфейса; ваша работа сводится к тому, чтобы просто переопределить два абстрактных метода.

Семантика методов Equals и GetHashCode подчиняется тем же правилам для методов object.Equals и object.GetHashCode, которые были описаны в главе 6. В следующем примере мы определяем класс Customer с двумя полями и затем записываем компаратор эквивалентности, сопоставляющий имена и фамилии:

```
public class Customer
{
    public string LastName;
    public string FirstName;

    public Customer (string last, string first)
    {
        LastName = last;
        FirstName = first;
    }
}

public class LastFirstEqComparer : EqualityComparer <Customer>
{
    public override bool Equals (Customer x, Customer y)
        => x.LastName == y.LastName && x.FirstName == y.FirstName;

    public override int GetHashCode (Customer obj)
        => (obj.LastName + ";" + obj.FirstName).GetHashCode();
}
```

Чтобы проиллюстрировать его работу, давайте создадим два экземпляра класса Customer:

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

Так как метод `object.Equals` не был переопределен, применяется нормальная семантика эквивалентности ссылочных типов:

```
Console.WriteLine (c1 == c2);           // False
Console.WriteLine (c1.Equals (c2));     // False
```

Та же самая стандартная семантика эквивалентности применяется, когда эти экземпляры используются в классе `Dictionary` без указания компаратора эквивалентности:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // False
```

А теперь укажем специальный компаратор эквивалентности:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // True
```

В приведенном примере необходимо проявлять осторожность, чтобы не изменить значение полей `FirstName` или `LastName` экземпляра `Customer` пока с ним производится работа в словаре, иначе изменится его хеш-код и функционирование словаря будет нарушено.

Свойство `EqualityComparer<T>.Default`

Свойство `EqualityComparer<T>.Default` возвращает универсальный компаратор эквивалентности, который может использоваться в качестве альтернативы вызову статического метода `object.Equals`. Его преимущество заключается в том, что такой компаратор сначала проверяет, реализует ли тип `T` интерфейс `IEquatable<T>`, и если реализует, то вызывает данную реализацию, избегая накладных расходов на упаковку. Это особенно удобно в обобщенных методах:

```
static bool Foo<T> (T x, T y)
{
    bool same = EqualityComparer<T>.Default.Equals (x, y);
    ...
}
```

Свойство `ReferenceEqualityComparer.Instance` (.NET 5+)

Начиная с .NET 5, свойство `ReferenceEqualityComparer.Instance` возвращает компаратор эквивалентности, который всегда применяет ссылочную эквивалентность. В случае типов значений его метод `Equals` всегда возвращает `false`.

Интерфейс `IComparer` и класс `Comparer`

Компараторы используются для переключения на специальную логику упорядочения в отсортированных словарях и коллекциях.

Обратите внимание, что компаратор бесполезен в несортированных словарях, таких как `Dictionary` и `Hashtable` — они требуют реализации `IEqualityComparer` для получения хеш-кодов. Подобным же образом компаратор эквивалентности бесполезен для отсортированных словарей и коллекций.

Ниже приведены определения интерфейса `IComparer`:

```
public interface IComparer
{
    int Compare(object x, object y);
}
public interface IComparer <in T>
{
    int Compare(T x, T y);
}
```

Как и в случае компараторов эквивалентности, имеется абстрактный класс, предназначенный для создания из него подклассов вместо реализации указанных интерфейсов:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
    public static Comparer<T> Default { get; }
    public abstract int Compare (T x, T y);           // Реализуется вами
    int IComparer.Compare (object x, object y);        // Реализован для вас
}
```

В следующем примере показан класс `Wish`, описывающий желание, и компаратор, который сортирует желания по приоритету:

```
class Wish
{
    public string Name;
    public int Priority;

    public Wish (string name, int priority)
    {
        Name = name;
        Priority = priority;
    }
}

class PriorityComparer : Comparer<Wish>
{
    public override int Compare (Wish x, Wish y)
    {
        if (object.Equals (x, y)) return 0;           // Оптимизация
        if (x == null) return -1;
        if (y == null) return 1;
        return x.Priority.CompareTo (y.Priority);
    }
}
```

Вызов метода `object.Equals` гарантирует, что путаница с методом `Equals` никогда не возникнет. В данном случае вызов статического метода `object.Equals` лучше вызова `x.Equals`, потому что он будет работать, даже если `x` имеет значение `null`!

Ниже показано, как применять класс PriorityComparer для сортировки содержимого List:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");
// ВЫВОД: 3 more wishes | Love | Peace | Wealth |
```

В следующем примере класс SurnameComparer позволяет сортировать строки фамилий в порядке, подходящем для телефонной книги:

```
class SurnameComparer : Comparer <string>
{
    string Normalize (string s)
    {
        s = s.Trim().ToUpper();
        if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
        return s;
    }

    public override int Compare (string x, string y)
        => Normalize (x).CompareTo (Normalize (y));
}
```

А вот как использовать класс SurnameComparer в отсортированном словаре:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
    Console.Write (s + " ");           // first! second! third!
```

Класс StringComparer

StringComparer — предопределенный подключаемый класс для поддержки эквивалентности и сравнения строк, который позволяет указывать язык и чувствительность к регистру символов. Он реализует интерфейсы IEqualityComparer и IComparer (плюс их обобщенные версии), так что может применяться с любым типом словаря или отсортированной коллекции.

Из-за того, что класс StringComparer является абстрактным, экземпляры получаются через его статические методы и свойства. Свойство StringComparer.Ordinal отражает стандартное поведение для строкового сравнения эквивалентности, а свойство StringComparer.CurrentCulture — стандартное поведение для сравнения порядка. Ниже перечислены все его статические члены:

```
public static StringComparer CurrentCulture { get; }
public static StringComparer CurrentCultureIgnoreCase { get; }
public static StringComparer InvariantCulture { get; }
public static StringComparer InvariantCultureIgnoreCase { get; }
public static StringComparer Ordinal { get; }
public static StringComparer OrdinalIgnoreCase { get; }
public static StringComparer Create (CultureInfo culture,
                                    bool ignoreCase);
```

В следующем примере создается ординальный нечувствительный к регистру словарь, в рамках которого dict["Joe"] и dict["JOE"] означают то же самое:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Вот как отсортировать массив имен с использованием австралийского английского:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

В финальном примере представлена учитываяющая культуру версия класса SurnameComparer, написанного в предыдущем разделе (со сравнением имен, подходящим для телефонной книги):

```
class SurnameComparer : Comparer<string>
{
    StringComparer strCmp;
    public SurnameComparer (CultureInfo ci)
    {
        // Создать строковый компаратор, чувствительный к регистру и культуре
        strCmp = StringComparer.Create (ci, false);
    }
    string Normalize (string s)
    {
        s = s.Trim();
        if (s.ToUpper ().StartsWith ("MC")) s = "MAC" + s.Substring (2);
        return s;
    }
    public override int Compare (string x, string y)
    {
        // Напрямую вызвать Compare на учитываящем культуру StringComparer
        return strCmp.Compare (Normalize (x), Normalize (y));
    }
}
```

Интерфейсы **IStructuralEquatable** и **IStructuralComparable**

Как обсуждалось в главе 6, по умолчанию структуры реализуют *структурное сравнение*: две структуры эквивалентны, если эквивалентны все их поля. Однако иногда структурная эквивалентность и сравнение порядка удобны в виде подключаемых вариантов также для других типов, таких как массивы и кортежи. В этом помогут следующие интерфейсы:

```
public interface IStructuralEquatable
{
    bool Equals (object other, IEqualityComparer comparer);
    int GetHashCode (IEqualityComparer comparer);
}

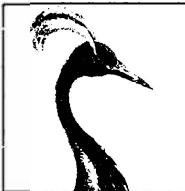
public interface IStructuralComparable
{
    int CompareTo (object other, IComparer comparer);
}
```

Передаваемая реализация `IEqualityComparer/IComparer` применяется к каждому индивидуальному элементу в составном объекте. Мы можем продемонстрировать это с использованием массивов и кортежей, которые реализуют указанные интерфейсы. В следующем примере мы сравниваем два массива на предмет эквивалентности сначала с применением стандартного метода `Equals`, а затем его версии из интерфейса `IStructuralEquatable`:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable sel1 = a1;
Console.WriteLine (a1.Equals (a2));                                // False
Console.WriteLine (sel1.Equals (a2, EqualityComparer<int>.Default)); // True
```

Вот еще один пример:

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
IStructuralEquatable sel1 = a1;
bool isTrue = sel1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

Запросы LINQ

Язык интегрированных запросов (Language Integrated Query — LINQ) представляет собой набор языковых средств и возможностей исполняющей среды, предназначенный для написания структурированных и безопасных в отношении типов запросов к локальным коллекциям объектов и удаленным источникам данных.

Язык LINQ позволяет создавать запросы к любой коллекции, которая реализует интерфейс `IEnumerable<T>`, будь то массив, список или DOM-модель XML, а также к удаленным источникам данных, таким как таблицы в базе данных SQL Server. Язык LINQ обладает преимуществами проверки типов на этапе компиляции и формирования динамических запросов.

В настоящей главе объясняется архитектура LINQ и фундаментальные основы написания запросов. Все основные типы определены в пространствах имен `System.Linq` и `System.Linq.Expressions`.



Примеры, рассматриваемые в текущей и двух последующих главах, доступны вместе с интерактивным инструментом запросов под названием LINQPad. Загрузить LINQPad можно на веб-сайте <http://www.linqpad.net>.

Начало работы

Базовыми единицами данных в LINQ являются *последовательности* и *элементы*. Последовательность — это любой объект, который реализует интерфейс `IEnumerable<T>`, а элемент — это каждая единица данных внутри последовательности. В следующем примере `names` является последовательностью, а `"Tom"`, `"Dick"` и `"Harry"` — элементами:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Мы называем ее *локальной последовательностью*, потому что она представляет локальную коллекцию объектов в памяти.

Операция запроса — это метод, который трансформирует последовательность. Типичная операция запроса принимает *входную последовательность* и выдает трансформированную *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операций запросов — все они реализованы в виде статических методов. Их называют *стандартными операциями запросов*.



Запросы, оперирующие на локальных последовательностях, называются *локальными запросами* или *запросами LINQ to Objects*.

Язык LINQ также поддерживает последовательности, которые могут динамически наполняться из удаленного источника данных, такого как база данных SQL Server. Последовательности подобного рода дополнительно реализуют интерфейс `IQueryable<T>` и поддерживаются через соответствующий набор стандартных операций запросов в классе `Queryable`. Мы обсудим данную тему более подробно в разделе “Интерпретируемые запросы” далее в главе.

Запрос представляет собой выражение, которое при перечислении трансформирует последовательности с помощью операций запросов. Простейший запрос состоит из одной входной последовательности и одной операции. Например, мы можем применить операцию `Where` к простому массиву для извлечения элементов с длиной, по меньшей мере, четырех символов:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where(
    names, n => n.Length >= 4);
foreach (string n in filteredNames)
    Console.WriteLine (n);
```

Вот как выглядит вывод:

```
Dick
Harry
```

Поскольку стандартные операции запросов реализованы в виде расширяющих методов, мы можем вызывать `Where` прямо на `names` — как если бы он был методом экземпляра:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Чтобы такой код скомпилировался, потребуется импортировать пространство имен `System.Linq`. Ниже приведен завершенный пример:

```
using System;
using System.Collections.Generic;
using System.Linq;
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
foreach (string name in filteredNames) Console.WriteLine (name);
```

Вывод будет таким:

```
Dick
Harry
```



Мы могли бы дополнительно сократить код, неявно типизируя `filteredNames`:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Однако в результате затрудняется зрительное восприятие запроса, особенно за пределами IDE-среды, где нет никаких всплывающих подсказок, которые помогли бы понять запрос. По этой причине в настоящей главе мы будем в меньшей степени использовать неявную типизацию, чем может оказаться у вас в собственном проекте.

Большинство операций запросов принимают в качестве аргумента лямбда-выражение. Лямбда-выражение помогает направлять и формировать запрос. В приведенном выше примере лямбда-выражение выглядит следующим образом:

```
n => n.Length >= 4
```

Входной аргумент соответствует входному элементу. В рассматриваемой ситуации входной аргумент `n` представляет каждое имя в массиве и относится к типу `string`. Операция `Where` требует, чтобы лямбда-выражение возвращало значение типа `bool`, которое в случае равенства `true` указывает на то, что элемент должен быть включен в выходную последовательность. Ниже приведена его сигнатура:

```
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, bool> predicate)
```

Следующий запрос извлекает все имена, которые содержат букву “`a`”:

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));
foreach (string name in filteredNames)
    Console.WriteLine (name);                                // Harry
```

До сих пор мы строили запросы с применением расширяющих методов и лямбда-выражений. Как вскоре вы увидите, такая стратегия хорошо компонуема в том смысле, что позволяет формировать цепочки операций запросов. В книге мы будем называть это *текучим синтаксисом*¹. Для написания запросов язык C# также предлагает другой синтаксис, который называется синтаксисом *выражений запросов*. Вот как записать предыдущий запрос в виде выражения запроса:

```
IEnumerable<string> filteredNames = from n in names
                                         where n.Contains ("a")
                                         select n;
```

Текущий синтаксис и синтаксис выражений запросов дополняют друг друга. В следующих двух разделах мы исследуем каждый из них более детально.

¹ Термин “текучий” (fluent) основан на работе Эрика Эванса и Мартина Фаулера, посвященной текучим интерфейсам.

Текущий синтаксис

Текущий синтаксис является наиболее гибким и фундаментальным. В этом разделе мы покажем, как выстраивать цепочки операций для формирования более сложных запросов, а также объясним важность расширяющих методов в данном процессе. Мы также расскажем, как формулировать лямбда-выражения для операции запроса, и представим несколько новых операций запросов.

Выстраивание в цепочки операций запросов

В предыдущем разделе были показаны два простых запроса, включающие одну операцию. Для построения более сложных запросов к выражению добавляются дополнительные операции запросов, формируя в итоге цепочку. В качестве примера следующий запрос извлекает все строки, содержащие букву “а”, сортирует их по длине и затем преобразует результаты в верхний регистр:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
    .Where (n => n.Contains ("a"))
    .OrderBy (n => n.Length)
    .Select (n => n.ToUpper ());

foreach (string name in query) Console.WriteLine (name);
```

Вывод:

```
JAY
MARY
HARRY
```



Переменная `n` в приведенном примере имеет закрытую область видимости в каждом лямбда-выражении. Идентификатор `n` можно многократно использовать по той же причине, по которой подобное возможно для идентификатора `s` в следующем методе:

```
void Test()
{
    foreach (char c in "string1") Console.Write (c);
    foreach (char c in "string2") Console.Write (c);
    foreach (char c in "string3") Console.Write (c);
}
```

`Where`, `OrderBy` и `Select` — стандартные операции запросов, которые распознаются как вызовы расширяющих методов класса `Enumerable` (если импортировано пространство имен `System.Linq`).

Мы уже представляли операцию `Where`, которая выдает отфильтрованную версию входной последовательности. Операция `OrderBy` выдает отсортированную версию своей входной последовательности, а метод `Select` — последова-

тельность, в которой каждый входной элемент трансформирован, или *спроецирован*, с помощью заданного лямбда-выражения (п.ToUpper в этом случае). Данные протекают слева направо через цепочку операций, так что они сначала фильтруются, затем сортируются и, наконец, проецируются.



Операция запроса никогда не изменяет входную последовательность; взамен она возвращает новую последовательность. Подход согласуется с парадигмой *функционального программирования*, которая была побудительной причиной создания языка LINQ.

Ниже приведены сигнатуры задействованных ранее расширяющих методов (с несколько упрощенной сигнатурой OrderBy):

```
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, bool> predicate)
public static IEnumerable<TSource> OrderBy<TSource, TKey>
    (this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

Когда операции запросов выстраиваются в цепочку, как в рассмотренном примере, выходная последовательность одной операции является входной последовательностью следующей операции. Полный запрос напоминает производственную линию с конвейерными лентами (рис. 8.1).

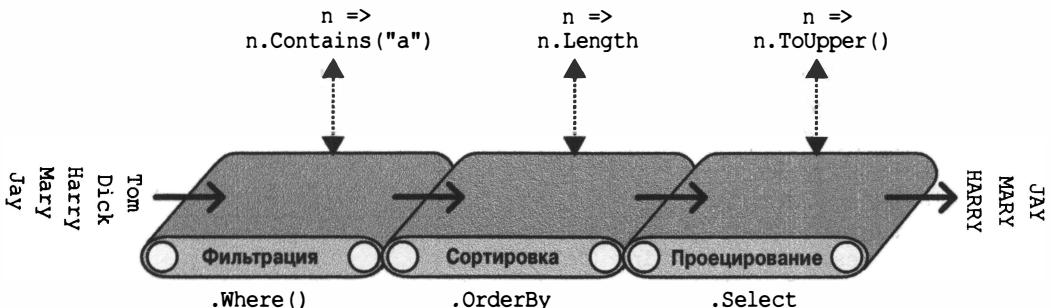


Рис. 8.1. Цепочка операций запросов

Точно такой же запрос можно строить *постепенно*, как показано ниже:

```
// Чтобы этот код скомпилировался, необходимо импортировать
// пространство имен System.Linq:
IEnumerable<string> filtered = names .Where (n => n.Contains ("a"));
IEnumerable<string> sorted = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted .Select (n => n.ToUpper());
```

Запрос finalQuery композиционно идентичен ранее сконструированному запросу query. Кроме того, каждый промежуточный шаг также состоит из допустимого запроса, который можно выполнить:

```
foreach (string name in filtered)
    Console.Write (name + "|");           // Harry|Mary|Jay|
Console.WriteLine();
foreach (string name in sorted)
    Console.Write (name + "|");           // Jay|Mary|Harry|
Console.WriteLine();
foreach (string name in finalQuery)
    Console.Write (name + "|");           // JAY|MARY|HARRY|
```

Почему расширяющие методы важны

Вместо применения синтаксиса расширяющих методов для вызова операций запросов можно использовать привычный синтаксис статических методов:

```
IEnumerable<string> filtered = Enumerable.Where (names, n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted, n => n.ToUpper());
```

Именно так компилятор фактически транслирует вызовы расширяющих методов. Однако избегание расширяющих методов не обходится даром, когда желательно записать запрос в одном операторе, как делалось ранее. Давайте вернемся к запросу в виде одного оператора — сначала с синтаксисом расширяющих методов:

```
IEnumerable<string> query = names.Where (n => n.Contains ("a"))
                                      .OrderBy (n => n.Length)
                                      .Select (n => n.ToUpper());
```

Его естественная линейная форма отражает протекание данных слева направо и также удерживает лямбда-выражения рядом с их операциями запросов (инфиксная система обозначений). Без расширяющих методов запрос утрачивает свою *текучесть*:

```
IEnumerable<string> query =
    Enumerable.Select (
        Enumerable.OrderBy (
            Enumerable.Where (
                names, n => n.Contains ("a")
            ), n => n.Length
        ), n => n.ToUpper()
    );
```

Составление лямбда-выражений

В предыдущем примере мы передаем операции *Where* следующее лямбда-выражение:

```
n => n.Contains ("a") // Входной тип - string, возвращаемый тип - bool
```



Лямбда-выражение, которое принимает значение и возвращает результат типа *bool*, называется *предикатом*.

Предназначение лямбда-выражения зависит от конкретной операции запроса. В операции *Where* оно указывает, должен ли элемент помещаться в вы-

ходную последовательность. В случае операции `OrderBy` лямбда-выражение отображает каждый элемент во входной последовательности на его ключ сортировки. В операции `Select` лямбда-выражение определяет, каким образом каждый элемент во входной последовательности трансформируется перед попаданием в выходную последовательность.



Лямбда-выражение в операции запроса всегда работает с индивидуальными элементами во входной последовательности, но не с последовательностью как единым целым.

Операция запроса вычисляет лямбда-выражение по требованию — обычно один раз на элемент во входной последовательности. Лямбда-выражения позволяют помещать внутрь операций запросов собственную логику. Это делает операции запросов универсальными и одновременно простыми по своей сути. Ниже приведена полная реализация `Enumerable.Where` кроме обработки исключений:

```
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, bool> predicate)
{
    foreach (TSource element in source)
        if (predicate (element))
            yield return element;
}
```

Лямбда-выражения и сигнатуры `Func`

Стандартные операции запросов задействуют обобщенные делегаты `Func`. Семейство универсальных обобщенных делегатов под названием `Func` определено в пространстве имен `System` со следующим замыслом.

Аргументы типа в `Func` появляются в том же самом порядке, что и в лямбда-выражениях.

Таким образом, `Func<TSource, bool>` соответствует лямбда-выражению `TSource=>bool`, которое принимает аргумент `TSource` и возвращает значение `bool`.

Аналогично `Func<TSource, TResult>` соответствует лямбда-выражению `TSource=>TResult`.

Делегаты `Func` перечислены в разделе “Делегаты `Func` и `Action`” главы 4.

Лямбда-выражения и типизация элементов

Стандартные операции запросов применяют описанные ниже имена обобщенных типов.

Имя обобщенного типа	Что означает
<code>TSource</code>	Тип элемента для входной последовательности
<code>TResult</code>	Тип элемента для выходной последовательности (если он отличается от <code>TSource</code>)
<code>TKey</code>	Тип элемента для ключа, используемого при сортировке, группировании или соединении

Тип `TSource` определяется входной последовательностью. Типы `TResult` и `TKey` обычно выводятся из лямбда-выражения.

Например, взгляните на сигнатуру операции запроса `Select`:

```
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

Делегат `Func<TSource, TResult>` соответствует лямбда-выражению `TSource=>TResult`, которое отображает входной элемент на выходной элемент. Типы `TSource` и `TResult` могут быть разными, так что лямбда-выражение способно изменять тип каждого элемента. Более того, лямбда-выражение определяет тип выходной последовательности. В следующем запросе с помощью операции `Select` выполняется трансформация элементов строкового типа в элементы целочисленного типа:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);
foreach (int length in query)
    Console.Write (length + "|");
                                // 3|4|5|4|3|
```

Компилятор может выводить тип `TResult` из возвращаемого значения лямбда-выражения. В данном случае `n.Length` возвращает значение `int`, поэтому для `TResult` выводится тип `int`. Операция запроса `Where` проще и не требует выведения типа для выходной последовательности, т.к. входной и выходной элементы относятся к тому же самому типу. Это имеет смысл, потому что данная операция просто фильтрует элементы; она не трансформирует их:

```
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func<TSource, bool> predicate)
```

Наконец, рассмотрим сигнатуру операции `OrderBy`:

```
// Сигнатура несколько упрощена:
public static IEnumerable<TSource> OrderBy<TSource, TKey>
    (this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)
```

Делегат `Func<TSource, TKey>` отображает входной элемент на ключ сортировки. Тип `TKey` выводится из лямбда-выражения и является отдельным от типов входного и выходного элементов. Например, можно реализовать сортировку списка имен по длине (ключ `int`) или в алфавитном порядке (ключ `string`):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength      = names.OrderBy (n => n.Length);           // ключ int
sortedAlphabetically = names.OrderBy (n => n);                  // ключ string
```



Операции запросов в классе `Enumerable` можно вызывать с помощью традиционных делегатов, ссылающихся на методы, а не на лямбда-выражения. Такой подход эффективен в плане упрощения некоторых видов локальных запросов — особенно LINQ to XML — и демонстрируется в главе 10. Однако он не работает с последовательностями, основанными на интерфейсе `IQueryable<T>` (например, при запрашивании базы данных), т.к. операции в классе `Queryable` требуют лямбда-выражений для выпуска деревьев выражений. Мы обсудим это позже в разделе “Интерпретируемые запросы”.

Естественный порядок

В языке LINQ исходный порядок элементов внутри входной последовательности важен. На такое поведение полагаются некоторые операции запросов, в частности, Take, Skip и Reverse.

Операция Take выдает первые x элементов, отбрасывая остальные:

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3); // { 10, 9, 8 }
```

Операция Skip игнорирует первые x элементов и выдает остальные:

```
IEnumerable<int> lastTwo = numbers.Skip (3); // { 7, 6 }
```

Операция Reverse изменяет порядок следования элементов на противоположный:

```
IEnumerable<int> reversed = numbers.Reverse(); // { 6, 7, 8, 9, 10 }
```

В локальных запросах (LINQ to Objects) операции наподобие Where и Select предохраняют исходный порядок во входной последовательности (как поступают все остальные операции запросов за исключением тех, которые специально изменяют порядок).

Другие операции

Не все операции запросов возвращают последовательность. Операции над элементами извлекают один элемент из входной последовательности; примерами таких операций служат First, Last, Single и ElementAt:

```
int[] numbers = { 10, 9, 8, 7, 6 };
int firstNumber = numbers.First(); // 10
int lastNumber = numbers.Last(); // 6
int secondNumber = numbers.ElementAt(1); // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First(); // 7
```

Операции агрегирования возвращают скалярное значение, обычно числового типа:

```
int count = numbers.Count(); // 5;
int min = numbers.Min(); // 6;
```

Квантификаторы возвращают значение bool:

```
bool hasTheNumberNine = numbers.Contains (9); // true
bool hasMoreThanZeroElements = numbers.Any(); // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0); // true
```

Поскольку эти операции возвращают одиночный элемент, вызов дополнительных операций запросов на их результате обычно не производится, если только сам элемент не является коллекцией.

Некоторые операции запросов принимают две входных последовательности. Примерами могут служить операция Concat, которая добавляет одну последовательность к другой, и операция Union, делающая то же самое, но с удалением дубликатов:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumarable<int> concat = seq1.Concat (seq2); // { 1, 2, 3, 3, 4, 5 }
IEnumarable<int> union = seq1.Union (seq2); // { 1, 2, 3, 4, 5 }
```

К данной категории относятся также и операции соединения. Все операции запросов подробно рассматриваются в главе 9.

Выражения запросов

Язык C# предоставляет синтаксическое сокращение для написания запросов LINQ, которое называется *выражениями запросов*. Вопреки распространенному мнению выражение запроса не является средством встраивания в C# возможностей языка SQL. В действительности на проектное решение для выражений запросов повлияли главным образом *выражения спискового включения* (они же *генераторы списков*; *list comprehension*) из таких языков функционального программирования, как LISP и Haskell, хотя косметическое влияние оказал и язык SQL.



В настоящей книге мысылаемся на синтаксис выражений запросов просто как на *синтаксис запросов*.

В предыдущем разделе с использованием текущего синтаксиса мы написали запрос для извлечения строк, содержащих букву "а", их сортировки и преобразования в верхний регистр. Ниже показано, как сделать то же самое с помощью синтаксиса запросов:

```
using System;
using System.Collections.Generic;
using System.Linq;

string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumarable<string> query =
    from n in names
    where n.Contains ("a") // Фильтровать элементы
    orderby n.Length        // Сортировать элементы
    select n.ToUpper(); // Транслировать (преобразовать) каждый элемент

foreach (string name in query) Console.WriteLine (name);
```

Вывод:

```
JAY
MARY
HARRY
```

Выражения запросов всегда начинаются с конструкции *from* и заканчиваются либо конструкцией *select*, либо конструкцией *group*. Конструкция *from* объявляет *переменную диапазона* (в данном случае *n*), которую можно воспринимать как переменную, предназначенную для обхода входной последовательности — почти как в цикле *foreach*. На рис. 8.2 представлен полный синтаксис в виде синтаксической (или т.н. железнодорожной) диаграммы.



Для чтения этой диаграммы начинайте слева и продолжайте двигаться по пути подобно поезду. Например, после обязательной конструкции `from` можно дополнительно включить конструкцию `orderby`, `where`, `let` или `join`. Затем можно либо продолжить конструкцией `select` или `group`, либо вернуться и включить еще одну конструкцию `from`, `orderby`, `where`, `let` или `join`.

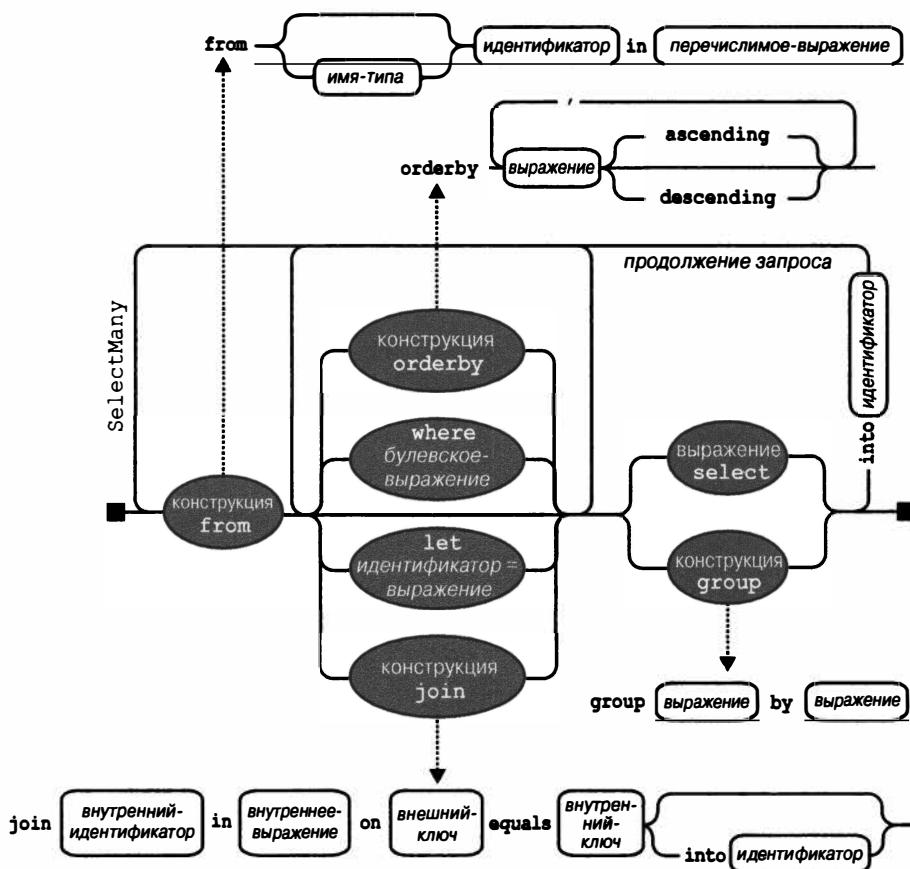


Рис. 8.2. Синтаксис запросов

Компилятор обрабатывает выражения запросов, транслируя их в текущий синтаксис. Трансляция делается в довольно-таки механической манере — очень похоже на то, как операторы `foreach` транслируются в вызовы методов `GetEnumerator` и `MoveNext`. Это значит, что любое выражение запроса, написанное с применением синтаксиса запросов, можно также представить посредством текущего синтаксиса. Компилятор (первоначально) транслирует показанный выше пример запроса в следующий код:

```
IEnumerable<string> query = names.Where (n => n.Contains ("a"))
    .OrderBy (n => n.Length)
    .Select (n => n.ToUpper());
```

Затем операции `Where`, `OrderBy` и `Select` преобразуются с использованием тех же правил, которые применялись бы к запросу, написанному с помощью текущего синтаксиса. В данном случае операции привязываются к расширяющим методам в классе `Enumerable`, т.к. пространство имен `System.Linq` импортировано и тип `names` реализует интерфейс `IEnumerable<string>`. Однако при трансляции выражений запросов компилятор не оказывает специальное содействие классу `Enumerable`. Можете считать, что компилятор механически вводит слова “`Where`”, “`OrderBy`” и “`Select`” внутрь оператора, после чего компилирует его, как если бы вы набирали такие имена методов самостоятельно. Это обеспечивает гибкость в том, как они распознаются. Например, операции в запросах к базе данных, которые мы будем строить в последующих разделах, взамен привязываются к расширяющим методам из класса `Queryable`.



Если удалить из программы директиву `using System.Linq`, тогда запросы не скомпилируются, поскольку методы `Where`, `OrderBy` и `Select` не к чему привязывать. Выражения запросов *не могут быть скомпилированы* до тех пор, пока не будет импортировано `System.Linq` или другое пространство имен с реализацией этих методов запросов.

Переменные диапазона

Идентификатор, непосредственно следующий за словом `from` в синтаксисе, называется *переменной диапазона*. Переменная диапазона ссылается на текущий элемент в последовательности, в отношении которой должна выполняться операция.

В наших примерах переменная диапазона `n` присутствует в каждой конструкции запроса. Однако в каждой конструкции эта переменная на самом деле выполняет *перечисление отличающейся последовательности*:

```
from      n in names          // n - переменная диапазона
where     n.Contains ("a")    // n берется прямо из массива
orderby   n.Length           // n впоследствии фильтруется
select    n.ToUpper()         // n впоследствии сортируется
```

Все станет ясным, если взглянуть на механическую трансляцию в текущий синтаксис, предпринимаемую компилятором:

```
names.Where (n => n.Contains ("a")) // n с локальной областью видимости
      .OrderBy (n => n.Length)        // n с локальной областью видимости
      .Select  (n => n.ToUpper())      // n с локальной областью видимости
```

Как видите, каждый экземпляр переменной `n` имеет закрытую область видимости, которая ограничена собственным лямбда-выражением.

Выражения запросов также позволяют вводить новые переменные диапазонов с помощью следующих конструкций:

- `let`
- `into`
- дополнительная конструкция `from`
- `join`

Мы рассмотрим данную тему позже в разделе “Стратегии композиции” настоящей главы и также в разделах “Выполнение проецирования” и “Выполнение соединения” главы 9.

Сравнение синтаксиса запросов и синтаксиса SQL

Выражения запросов внешне похожи на код SQL, хотя они существенно отличаются. Запрос LINQ сводится к выражению C#, а потому следует стандартным правилам языка C#. Например, в LINQ нельзя использовать переменную до ее объявления. Язык SQL разрешает ссылаться в операторе SELECT на псевдоним таблицы до его определения в конструкции FROM.

Подзапрос в LINQ является просто еще одним выражением C#, так что никакого специального синтаксиса он не требует. Подзапросы в SQL подчиняются специальным правилам.

В LINQ данные логически протекают слева направо через запрос. Что касается потока данных в SQL, то порядок структурирован не настолько хорошо.

Запрос LINQ состоит из конвейера операций, принимающих и выпускающих последовательности, в которых порядок следования элементов может иметь значение. Запрос SQL образован из сети конструкций, которые работают по большей части с неупорядоченными наборами.

Сравнение синтаксиса запросов и текущего синтаксиса

И синтаксис выражений запросов, и текущий синтаксис обладают своими преимуществами.

Синтаксис запросов проще для запросов, которые содержат в себе любой из перечисленных ниже аспектов:

- конструкцию let для введения новой переменной наряду с переменной диапазона;
- операцию SelectMany, Join или GroupJoin, за которой следует ссылка на внешнюю переменную диапазона.

(Мы опишем конструкцию let в разделе “Стратегии композиции” далее в главе, а операции SelectMany, Join и GroupJoin — в главе 9.)

Посредине находятся запросы, которые просто применяют операции Where, OrderBy и Select. С ними одинаково хорошо работает любой синтаксис; выбор здесь определяется в основном личными предпочтениями.

Для запросов, состоящих из одной операции, текущий синтаксис короче и характеризуется меньшим беспорядком.

Наконец, есть много операций, для которых ключевые слова в синтаксисте запросов не предусмотрены. Такие операции требуют использования текущего синтаксиса — по крайней мере, частично. К ним относится любая операция, выходящая за рамки перечисленных ниже:

Where, Select, SelectMany,
OrderBy, ThenBy, OrderByDescending, ThenByDescending,
GroupBy, Join, GroupJoin

Запросы со смешанным синтаксисом

Если операция запроса не поддерживается в синтаксисе запросов, тогда синтаксис запросов и текущий синтаксис можно смешивать. Единственное ограничение заключается в том, что каждый компонент синтаксиса запросов должен быть завершен (т.е. начинаться с конструкции `from` и заканчиваться конструкцией `select` или `group`).

Предположим, что есть такое объявление массива:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

В следующем примере подсчитывается количество имен, содержащих букву “`a`”:

```
int matches = (from n in names where n.Contains ("a") select n).Count(); //3
```

Показанный ниже запрос получает имя, которое находится первым в алфавитном порядке:

```
string first = (from n in names orderby n select n).First(); // Dick
```

Подход со смешанным синтаксисом иногда полезен в более сложных запросах. Однако в представленных выше простых примерах мы могли бы безо всяких проблем придерживаться текущего синтаксиса:

```
int matches = names.Where (n => n.Contains ("a")).Count(); // 3  
string first = names.OrderBy (n => n).First(); // Dick
```



Временами запросы со смешанным синтаксисом обеспечивают почти максимальную выгоду в плане функциональности и простоты. Важно не отдавать одностороннее предпочтение синтаксису запросов или текущему синтаксису, иначе вы не сможете записывать запросы со смешанным синтаксисом, когда они являются наилучшим вариантом.

В оставшейся части главы мы будем демонстрировать ключевые концепции с применением обоих видов синтаксиса, когда это уместно.

Отложенное выполнение

Важная особенность большинства операций запросов связана с тем, что они выполняются не тогда, когда создаются, а когда происходит *перечисление* (другими словами, при вызове метода `MoveNext` на перечислителе). Рассмотрим следующий запрос:

```
var numbers = new List<int>();  
numbers.Add (1);  
  
IEnumerable<int> query = numbers.Select (n => n * 10); // Построить запрос  
numbers.Add (2); // Вставить дополнительный элемент  
  
foreach (int n in query)  
    Console.Write (n + "|"); // 10|20|
```

Дополнительное число, вставленное в список *после* конструирования запроса, включается в результат, поскольку любая фильтрация или сортировка не делается вплоть до выполнения оператора `foreach`. Это называется *отложенным* или *ленивым* выполнением и представляет собой то же самое действие, которое происходит с делегатами:

```
Action a = () => Console.WriteLine ("Foo");
// Пока на консоль ничего не выводится. А теперь запустим запрос:
a(); // Отложенное выполнение!
```

Отложенное выполнение поддерживают все стандартные операции запросов со следующими исключениями:

- операции, которые возвращают одиночный элемент или скалярное значение, такие как `First` или `Count`;
- перечисленные ниже *операции преобразования*:

`ToArray`, `ToList`, `ToDictionary`, `ToLookup`, `ToHashSet`

Указанные операции вызывают немедленное выполнение запроса, т.к. их результирующие типы не имеют механизма для обеспечения отложенного выполнения. Скажем, метод `Count` возвращает простое целочисленное значение, для которого последующее перечисление невозможно. Показанный ниже запрос выполняется немедленно:

```
int matches = numbers.Where (n => n <= 2).Count(); // 1
```

Отложенное выполнение важно из-за того, что оно *отвязывает конструирование* запроса от его *выполнения*. Это позволяет строить запрос в течение нескольких шагов и также делает возможными запросы к базе данных.



Подзапросы предоставляют еще один уровень косвенности. Все, что находится в подзапросе, подпадает под отложенное выполнение — включая методы агрегирования и преобразования. Мы рассмотрим их в разделе “Подзапросы” далее в главе.

Повторное вычисление

С отложенным выполнением связано еще одно последствие — запрос с отложенным выполнением при повторном перечислении вычисляется заново:

```
var numbers = new List<int>() { 1, 2 };
IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write (n + "|"); // 10|20|
numbers.Clear();
foreach (int n in query) Console.Write (n + "|"); // Ничего не выводится
```

Есть пара причин, по которым повторное вычисление иногда неблагоприятно:

- временами требуется “заморозить” или кэшировать результаты в определенный момент времени;
- некоторые запросы сопровождаются большим объемом вычислений (или полагаются на обращение к удаленной базе данных), поэтому повторять их без настоятельной необходимости нежелательно.

Повторного вычисления можно избежать за счет вызова операции преобразования, такой как `ToArray` или `ToList`. Операция `ToArray` копирует выходные данные запроса в массив, а `ToList` — в обобщенный список `List<T>`:

```
var numbers = new List<int>() { 1, 2 };
List<int> timesTen = numbers
    .Select (n => n * 10)
    .ToList(); // Выполняется немедленное преобразование в List<int>
numbers.Clear();
Console.WriteLine (timesTen.Count); // По-прежнему 2
```

Захваченные переменные

Если лямбда-выражения запроса *захватывают* внешние переменные, то запрос будет принимать на обработку значения таких переменных в момент, когда он запускается:

```
int[] numbers = { 1, 2 };
int factor = 10;
IQueryable<int> query = numbers.Select (n => n * factor);
factor = 20;
foreach (int n in query) Console.Write (n + "|"); // 20|40|
```

В итоге может возникнуть проблема при построении запроса внутри цикла `for`. Например, предположим, что необходимо удалить все гласные из строки. Следующий код, несмотря на свою неэффективность, дает корректный результат:

```
IEnumerable<char> query = "Not what you might expect";
query = query.Where (c => c != 'a');
query = query.Where (c => c != 'e');
query = query.Where (c => c != 'i');
query = query.Where (c => c != 'o');
query = query.Where (c => c != 'u');
foreach (char c in query) Console.Write (c); // Nt wht y mght xpct
```

А теперь посмотрим, что произойдет, если мы переделаем код с использованием цикла `for`:

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";
for (int i = 0; i < vowels.Length; i++)
    query = query.Where (c => c != vowels[i]);
foreach (char c in query) Console.Write (c);
```

При перечислении запроса генерируется исключение `IndexOutOfRangeException`, потому что, как было указано в разделе “Захватывание внешних переменных” главы 4, компилятор назначает переменной итерации в цикле `for` такую же область видимости, как если бы она была объявлена *вне* цикла. Следовательно, каждое замыкание захватывает *ту же самую* переменную (`i`), значение которой равно 5, когда начинается действительное перечисление запроса. Чтобы решить проблему, переменную цикла потребуется присвоить другой переменной, объявленной *внутри* блока операторов:

```
for (int i = 0; i < vowels.Length; i++)
{
    char vowel = vowels[i];
    query = query.Where (c => c != vowel);
}
```

В таком случае на каждой итерации цикла будет захватываться свежая локальная переменная.



Еще один способ решения описанной проблемы предусматривает замену цикла `for` циклом `foreach`:

```
foreach (char vowel in vowels)
    query = query.Where (c => c != vowel);
```

Как работает отложенное выполнение

Операции запросов обеспечивают отложенное выполнение за счет возвращения *декораторных* последовательностей.

В отличие от традиционного класса коллекции, такого как массив или связанный список, декораторная последовательность (в общем случае) не имеет собственной поддерживающей структуры для хранения элементов. Взамен она является оболочкой для другой последовательности, предоставляемой во время выполнения, и поддерживает с ней постоянную зависимость. Всякий раз, когда запрашиваются данные из декоратора, он в свою очередь должен запрашивать данные из внутренней входной последовательности.



Трансформация операции запроса образует “декорацию”. Если выходная последовательность не подвергается трансформациям, то результатом будет простой *посредник*, а не декоратор.

Вызов операции `Where` просто конструирует декораторную последовательность-оболочку, которая хранит ссылку на входную последовательность, лямбда-выражение и любые другие указанные аргументы. Входная последовательность перечисляется только при перечислении декоратора.

На рис. 8.3 проиллюстрирована композиция следующего запроса:

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }.Where (n => n < 10);
```

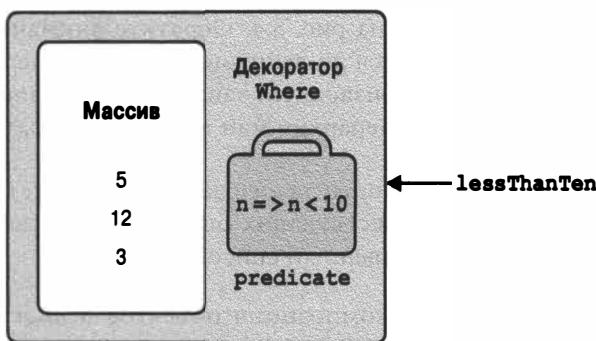


Рис. 8.3. Декораторная последовательность

При перечислении lessThanTen в действительности происходит запрос массива через декоратор Where.

Хорошая новость заключается в том, что даже если требуется создать собственную операцию запроса, то декораторная последовательность легко реализуется с помощью итератора C#. Ниже показано, как можно написать собственный метод MySelect:

```
public static IEnumerable<TResult> MySelect<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    foreach (TSource element in source) yield return selector (element);
}
```

Данный метод является итератором благодаря наличию оператора `yield return`. С точки зрения функциональности он представляет собой сокращение для следующего кода:

```
public static IEnumerable<TResult> MySelect<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    return new SelectSequence (source, selector);
}
```

где `SelectSequence` — это (сгенерированный компилятором) класс, перечислитель которого инкапсулирует логику из метода итератора.

Таким образом, при вызове операции вроде `Select` или `Where` всего лишь создается экземпляр перечислимого класса, который декорирует входную последовательность.

Построение цепочки декораторов

Объединение операций запросов в цепочку приводит к созданию иерархических представлений декораторов. Рассмотрим следующий запрос:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where (n => n < 10)
    .OrderBy (n => n)
    .Select (n => n * 10);
```

Каждая операция запроса создает новый экземпляр декоратора, который является оболочкой для предыдущей последовательности (подобно матрешке). Объектная модель этого запроса показана на рис. 8.4. Обратите внимание, что объектная модель полностью конструируется до выполнения любого перечисления.

При перечислении `query` производятся запросы к исходному массиву, трансформированному посредством иерархии или цепочки декораторов.



Добавление `ToList` в конец такого запроса приведет к немедленному выполнению предшествующих операций, что свернет всю объектную модель в единственный список.

На рис. 8.5 показана та же композиция объектов в виде диаграммы UML (Unified Modeling Language — универсальный язык моделирования).

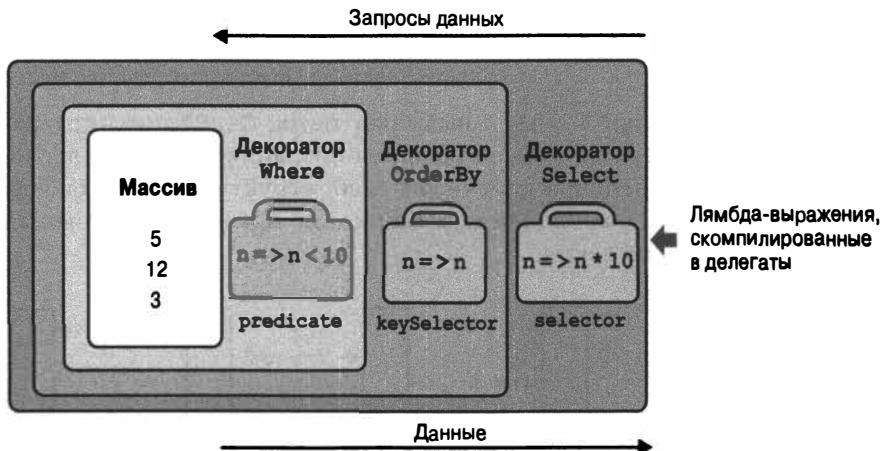


Рис. 8.4. Иерархия декораторных последовательностей

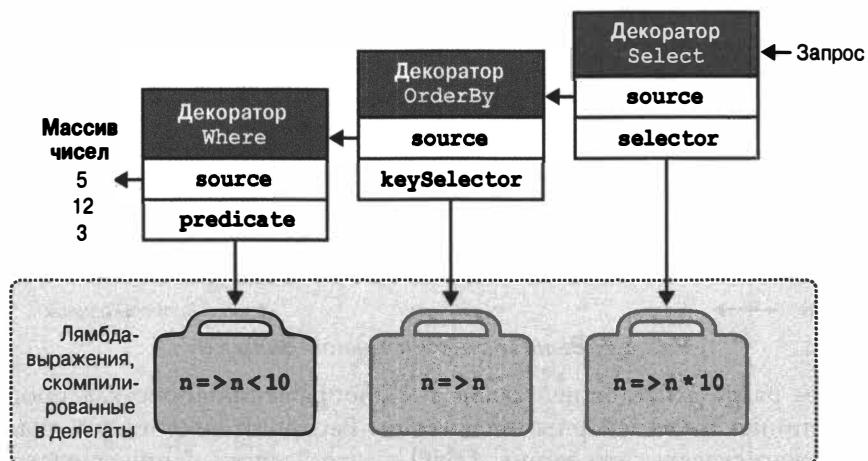


Рис. 8.5. UML-диаграмма композиции декораторов

Декоратор Select ссылается на декоратор OrderBy, который в свою очередь ссылается на декоратор Where, а тот — на массив. Особенность отложенного выполнения заключается в том, что при постепенном формировании запроса строится идентичная объектная модель:

```
IEnumerable<int>
source = new int[] 5, 12, 3 ,
filtered = source .Where (n => n < 10),
sorted = filtered .OrderBy (n => n),
query = sorted .Select (n => n * 10);
```

Каким образом выполняются запросы

Ниже представлены результаты перечисления предыдущего запроса:

```
foreach (int n in query) Console.WriteLine (n);
```

Вот вывод:

30

50

“За кулисами” цикл `foreach` вызывает метод `GetEnumerator` на декораторе `Select` (последняя или самая внешняя операция), который все и запускает. Результатом будет цепочка перечислителей, структурно отражающая цепочку декораторных последовательностей. На рис. 8.6 показан поток выполнения при прохождении перечисления.

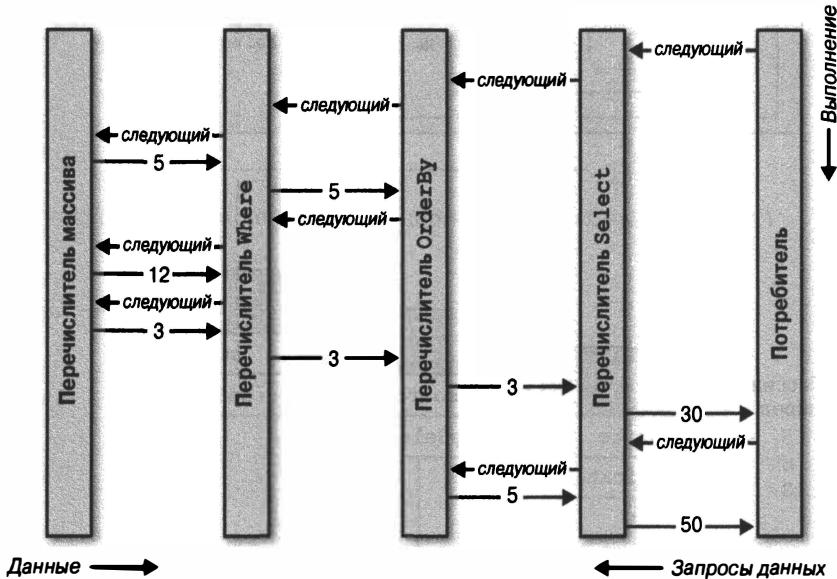


Рис. 8.6. Выполнение локального запроса

В первом разделе настоящей главы мы изображали запрос как производственную линию с конвейерными лентами. Распространяя такую аналогию дальше, можно сказать, что запрос LINQ — это “ленивая” производственная линия, в которой конвейерные ленты перемещают элементы только по *требованию*. Построение запроса конструирует производственную линию со всеми составными частями на своих местах, но в остановленном состоянии. Когда потребитель запрашивает элемент (выполняет перечисление запроса), активизируется самая правая конвейерная лента; такое действие в свою очередь запускает остальные конвейерные ленты — когда требуются элементы входной последовательности. Язык LINQ следует модели с *пассивным источником*, управляемой запросом, а не модели с *активным источником*, управляемой подачей. Это важный аспект, который, как будет показано далее, позволяет распространить LINQ на выдачу запросов к базам данных SQL.

Подзапросы

Подзапрос представляет собой запрос, содержащийся внутри лямбда-выражения другого запроса.

В следующем примере подзапрос применяется для сортировки музыкантов по фамилии:

```
string[] musos =  
    { "David Gilmour", "Roger Waters", "Rick Wright", "Nick Mason" };  
IEnumerable<string> query = musos.OrderBy(m => m.Split().Last());
```

Вызов `m.Split` преобразует каждую строку в коллекцию слов, на которой затем вызывается операция запроса `Last`. Здесь `m.Split().Last()` является подзапросом, а `query` — *внешним запросом*.

Подзапросы разрешены, т.к. с правой стороны лямбда-выражения можно помещать любое допустимое выражение C#. Подзапрос — это просто еще одно выражение C#. Таким образом, правила для подзапросов будут следствием из правил для лямбда-выражений (и общего поведения операций запросов).



В общем смысле термин “подзапрос” имеет более широкое значение. При описании LINQ мы используем данный термин только для запроса, находящегося внутри лямбда-выражения другого запроса. В выражении запроса подзапрос означает запрос, на который производится ссылка из выражения в любой конструкции кроме `from`.

Подзапрос имеет закрытую область видимости внутри включающего выражения и способен ссылаться на параметры во внешнем лямбда-выражении (или на переменные диапазона в выражении запроса).

Конструкция `m.Split().Last` — очень простой подзапрос. Следующий запрос извлекает из массива самые короткие строки:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };  
IEnumerable<string> outerQuery = names  
    .Where(n => n.Length == names.OrderBy(n2 => n2.Length)  
        .Select(n2 => n2.Length).First());
```

ВЫВОД:

Tom, Jay

А вот как получить то же самое с помощью выражения запроса:

```
IEnumerable<string> outerQuery =  
    from n in names  
    where n.Length ==  
        (from n2 in names orderby n2.Length select n2.Length).First()  
    select n;
```

Поскольку внешняя переменная диапазона (`n`) находится в области видимости подзапроса, применять `n` в качестве переменной диапазона подзапроса нельзя.

Подзапрос выполняется каждый раз, когда вычисляется включающее его лямбда-выражение. Это значит, что подзапрос выполняется по требованию, на усмотрение внешнего запроса. Можно было бы сказать, что процесс выполнения продвигается *снаружи внутрь*. Локальные запросы следуют такой модели буквально, а интерпретируемые запросы (например, запросы к базе данных) следуют ей *концептуально*.

Подзапрос выполняется, когда это требуется для передачи данных внешнему запросу. Как иллюстрируется на рис. 8.7 и 8.8, в рассматриваемом примере подзапрос (верхняя конвейерная лента на рис. 8.7) выполняется один раз для каждой итерации внешнего цикла.

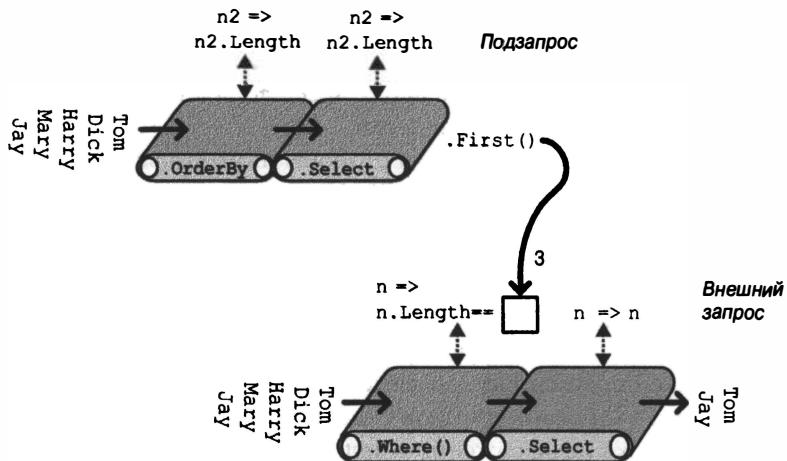


Рис. 8.7. Композиция подзапроса

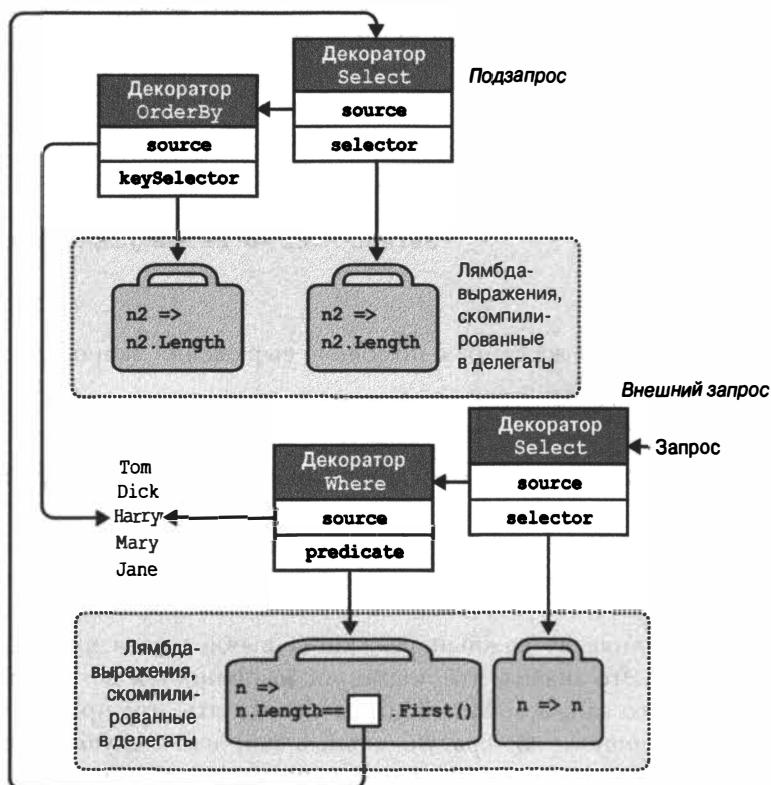


Рис. 8.8. UML-диаграмма композиции подзапроса

Предыдущий подзапрос можно выразить более лаконично:

```
IEnumerable<string> query =
    from n in names
    where n.Length == names.OrderBy (n2 => n2.Length).First().Length
    select n;
```

С помощью функции агрегирования Min запрос можно еще больше упростить:

```
IEnumerable<string> query =
    from n in names
    where n.Length == names.Min (n2 => n2.Length)
    select n;
```

В разделе “Интерпретируемые запросы” далее в главе мы покажем, каким образом отправлять запросы к удаленным источникам данных, таким как таблицы SQL. В нашем примере делается идеальный запрос к базе данных, потому что он может быть обработан как единое целое, требуя только одного обращения к серверу базы данных. Однако этот запрос неэффективен для локальной коллекции, т.к. на каждой итерации внешнего цикла подзапрос вычисляется повторно. Подобной неэффективности можно избежать, запуская подзапрос отдельно (так что он перестает быть подзапросом):

```
int shortest = names.Min (n => n.Length);
IQueryable<string> query = from n in names
                             where n.Length == shortest
                             select n;
```



Вынесение подзапросов в подобном стиле почти всегда желательно при выполнении запросов к локальным коллекциям. Исключение — ситуация, когда подзапрос является *коррелированным*, т.е. ссылается на внешнюю переменную диапазона. Коррелированные подзапросы рассматриваются в разделе “Выполнение проецирования” главы 9.

Подзапросы и отложенное выполнение

Наличие в подзапросе операции над элементами или операции агрегирования, такой как First или Count, не приводит к немедленному выполнению внешнего запроса — для внешнего запроса по-прежнему поддерживается отложенное выполнение. Причина в том, что подзапросы вызываются *косвенно* — через делегат в случае локального запроса или через дерево выражения в случае интерпретируемого запроса.

Интересная ситуация возникает при помещении подзапроса внутрь выражения Select. Для локального запроса фактически *производится проецирование последовательности запросов*, каждый из которых подпадает под отложенное выполнение. Результат обычно прозрачен и служит для дальнейшего улучшения эффективности. Подзапросы Select еще будут рассматриваться в главе 9.

Стратегии композиции

В настоящем разделе мы опишем три стратегии для построения более сложных запросов:

- постепенное построение запросов;
- использование ключевого слова `into`;
- упаковка запросов.

Все они являются стратегиями *выстраивания в цепочки* и во время выполнения выдают идентичные запросы.

Постепенное построение запросов

В начале главы мы демонстрировали, что текущий запрос можно было бы строить постепенно:

```
var filtered = names .Where (n => n.Contains ("a"));
var sorted = filtered .OrderBy (n => n);
var query = sorted .Select (n => n.ToUpper());
```

Из-за того, что каждая участвующая операция запроса возвращает декораторную последовательность, результирующим запросом будет та же самая цепочка либо иерархия декораторов, которая была бы получена из запроса с единственным выражением. Тем не менее, постепенное построение запросов обладает парой потенциальных преимуществ.

- Оно может упростить написание запросов.
- Операции запросов можно добавлять *условно*. Например, следующий прием:

```
if (includeFilter) query = query.Where (...)
```

более эффективен, чем такой вариант:

```
query = query.Where (n => !includeFilter || <выражение>)
```

поскольку позволяет избежать добавления дополнительной операции запроса, если `includeFilter` равно `false`.

Постепенный подход часто полезен в плане охвата запросов. В целях иллюстрации предположим, что нужно удалить все гласные из списка имен и затем представить в алфавитном порядке те из них, длина которых все еще превышает два символа. С помощью текущего синтаксиса мы могли бы записать такой запрос в форме единственного выражения, произведя проецирование *перед* фильтрацией:

```
IEnumerable<string> query = names
    .Select (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", ""))
        .Replace ("o", "").Replace ("u", ""))
    .Where (n => n.Length > 2)
    .OrderBy (n => n);

// Dck
// Hrry
// Mry
```



Вместо того чтобы вызывать метод Replace типа string пять раз, мы могли бы удалить гласные из строки более эффективно посредством регулярного выражения:

```
n => Regex.Replace (n, "[aeiou]", "")
```

Однако метод Replace типа string обладает тем преимуществом, что работает также в запросах к базам данных.

Трансляция такого кода напрямую в выражение запроса довольно непроста, потому что конструкция select должна находиться после конструкций where и orderby. А если переупорядочить запрос так, чтобы проецирование выполнялось последним, то результат будет другим:

```
IEnumerable<string> query =
    from n in names
    where n.Length > 2
    orderby n
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
        .Replace ("o", "").Replace ("u", "");

// Dck
// Hrry
// Jy
// Mry
// Tm
```

К счастью, существует несколько способов получить первоначальный результат с помощью синтаксиса запросов. Первый из них — постепенное формирование запроса:

```
IEnumerable<string> query =
    from n in names
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
        .Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
// Dck
// Hrry
// Mry
```

Ключевое слово `into`



В зависимости от контекста ключевое слово `into` интерпретируется выражениями запросов двумя совершенно разными путями. Его первое предназначение, которое мы опишем здесь — сигнализация о *продолжении запроса* (другим предназначением является сигнализация о `GroupJoin`).

Ключевое слово `into` позволяет “продолжить” запрос после проецирования и является сокращением для постепенного построения запросов.

Предыдущий запрос можно переписать с применением `into`:

```
IEnumerable<string> query =
    from n in names
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
        .Replace ("o", "").Replace ("u", "")
    into noVowel
    where noVowel.Length > 2 orderby noVowel select noVowel;
```

Единственное место, где можно использовать `into` — после конструкции `select` или `group`. Ключевое слово `into` “начинает заново” запрос, позволяя вводить новые конструкции `where`, `orderby` и `select`.



Хотя с точки зрения выражения запроса ключевое слово `into` проще считать средством начать запрос заново, после трансляции в финальную текущую форму все становится **одним запросом**. Следовательно, ключевое слово `into` не привносит никаких дополнительных расходов в плане производительности. Применяя его, вы совершенно ничего не теряете!

Эквивалентом `into` в текущем синтаксисе является просто более длинная цепочка операций.

Правила области видимости

После ключевого слова `into` все переменные диапазона покидают область видимости. Следующий код не скомпилируется:

```
var query =
    from n1 in names
    select n1.ToUpper()
    into n2           // Начиная с этого места, видна только переменная n2
    where n1.Contains ("x")      // Недопустимо: n1 не находится
                                    // в области видимости
    select n2;
```

Чтобы понять причину, давайте посмотрим, как показанный код отображается на текущий синтаксис:

```
var query = names
    .Select (n1 => n1.ToUpper())
    .Where (n2 => n1.Contains ("x")); // Ошибка: переменная n1 не
                                    // находится в области видимости
```

К тому времени, когда запускается фильтр `Where`, исходная переменная (`n1`) уже утрачена. Входная последовательность `Where` содержит только имена в верхнем регистре, и ее фильтрация на основе `n1` невозможна.

Упаковка запросов

Запрос, построенный постепенно, может быть сформулирован как единственный оператор за счет упаковки одного запроса в другой.

В общем случае запрос:

```
var tempQuery = tempQueryExpr  
var finalQuery = from ... in tempQuery ...
```

можно переформулировать следующим образом:

```
var finalQuery = from ... in (tempQueryExpr)
```

Упаковка семантически идентична постепенному построению запросов либо использованию ключевого слова `into` (без промежуточной переменной). Во всех случаях конечным результатом будет линейная цепочка операций запросов. Например, рассмотрим показанный ниже запрос:

```
IEnumerable<string> query =  
    from n in names  
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")  
        .Replace ("o", "").Replace ("u", "");  
query = from n in query where n.Length > 2 orderby n select n;
```

Вот как он выглядит, когда переведен в упакованную форму:

```
IEnumerable<string> query =  
    from n1 in  
    (  
        from n2 in names  
        select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")  
            .Replace ("o", "").Replace ("u", "")  
    )  
    where n1.Length > 2 orderby n1 select n1;
```

После преобразования в текущий синтаксис в результате получается та же самая линейная цепочка операций, что и в предшествующих примерах:

```
IEnumerable<string> query = names  
    .Select (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")  
        .Replace ("o", "").Replace ("u", ""))  
    .Where (n => n.Length > 2)  
    .OrderBy (n => n);
```

(Компилятор не выпускает финальный вызов `Select (n => n)`, т.к. он избыточен.)

Упакованные запросы могут несколько запутывать, поскольку они имеют сходство с подзапросами, которые рассматривались ранее. Обе разновидности поддерживают концепции внутреннего и внешнего запросов. Однако при преобразовании в текущий синтаксис можно заметить, что упаковка — это просто стратегия для последовательного выстраивания операций в цепочку. Конечный результат совершенно не похож на подзапрос, который встраивает внутренний запрос в лямбда-выражение другого запроса.

Возвращаясь к ранее примененной аналогии: при упаковке “внутренний” запрос имитирует *предшествующую конвейерную ленту*. И напротив, подзапрос перемещается по конвейерной ленте и активизируется по требованию посредством “ламбда-рабочего” конвейерной ленты (см. рис. 8.7).

Стратегии проецирования

Инициализаторы объектов

До сих пор все наши конструкции `select` проецировали в скалярные типы элементов. С помощью инициализаторов объектов C# можно выполнять проецирование в более сложные типы. Например, предположим, что в качестве первого шага запроса мы хотим удалить гласные из списка имен, одновременно сохраняя рядом исходные версии для последующих запросов. В помощь этому мы можем написать следующий класс:

```
class TempProjectionItem
{
    public string Original; // Исходное имя
    public string Vowelless; // Имя с удаленными гласными
}
```

Затем мы можем проецировать в него посредством инициализаторов объектов:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<TempProjectionItem> temp =
    from n in names
    select new TempProjectionItem
    {
        Original = n,
        Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                    .Replace ("o", "").Replace ("u", "")
    };
```

Результатом будет тип `IEnumerable<TempProjectionItem>`, которому впоследствии можно отправлять запросы:

```
IEnumerable<string> query = from item in temp
                             where item.Vowelless.Length > 2
                             select item.Original;

// Dick
// Harry
// Mary
```

Анонимные типы

Анонимные типы позволяют структурировать промежуточные результаты без написания специальных классов. С помощью анонимных типов в предыдущем примере можно избавиться от класса `TempProjectionItem`:

```
var intermediate = from n in names
    select new
    {
        Original = n,
        Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                     .Replace ("o", "").Replace ("u", "")
    };
```

```
IEnumerable<string> query = from item in intermediate
    where item.Vowelless.Length > 2
    select item.Original;
```

Результат будет таким же, как в предыдущем примере, но без необходимости в написании одноразового класса. Всю нужную работу проделает компилятор, сгенерировав класс с полями, которые соответствуют структуре нашей проекции. Тем не менее, это означает, что запрос `intermediate` имеет следующий тип:

```
IEnumerable<случайное-имя-сгенерированное-компилятором>
```

Единственный способ объявления переменной такого типа предусматривает использование ключевого слова `var`. В данном случае `var` является не просто средством сокращения беспорядка, а настоятельной необходимостью.

С применением ключевого слова `into` запрос можно записать более лаконично:

```
var query = from n in names
    select new
    {
        Original = n,
        Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
            .Replace ("o", "").Replace ("u", "")
    }
    into temp
    where temp.Vowelless.Length > 2
    select temp.Original;
```

Выражения запросов предлагают сокращение для написания запросов подобного вида — ключевое слово `let`.

Ключевое слово `let`

Ключевое слово `let` вводит новую переменную параллельно переменной диапазона.

Написать запрос, извлекающий строки, длина которых после исключения гласных превышает два символа, посредством `let` можно так:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IQueryable<string> query =
    from n in names
    let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
        .Replace ("o", "").Replace ("u", "")
    where vowelless.Length > 2
    orderby vowelless
    select n;      // Благодаря let переменная n по-прежнему находится
                    // в области видимости
```

Компилятор распознает конструкцию `let` путем проецирования во временный анонимный тип, который содержит переменную диапазона и новую переменную выражения. Другими словами, компилятор транслирует этот запрос в показанный ранее пример.

Ключевое слово let решает две задачи:

- оно проецирует новые элементы наряду с существующими элементами;
- оно позволяет многократно использовать выражение в запросе, не записывая его каждый раз заново.

В приведенном примере подход с `let` особенно полезен, т.к. он позволяет конструкции `select` выполнять проецирование либо исходного имени (`n`), либо его версии с удаленными гласными (`vowelless`).

Можно иметь любое количество конструкций `let`, находящихся до или после `where` (см. рис. 8.2). В операторе `let` можно ссылаться на переменные, введенные в более ранних операторах `let` (в зависимости от границ, наложенных конструкцией `into`). Оператор `let` *повторно проецирует* все существующие переменные прозрачным образом.

Выражение `let` не должно вычисляться как значение скалярного типа: его иногда полезно вычислять, скажем, в подпоследовательность.

Интерпретируемые запросы

Язык LINQ параллельно поддерживает две архитектуры: *локальные* запросы для локальных коллекций объектов и *интерпретируемые* запросы для удаленных источников данных. До сих пор мы исследовали архитектуру локальных запросов, которые действуют на коллекциях, реализующих интерфейс `IEnumerable<T>`. Локальные запросы преобразуются в операции запросов из класса `Enumerable` (по умолчанию), которые в свою очередь распознаются как цепочки декораторных последовательностей. Делегаты, которые они принимают — выраженные с применением синтаксиса запросов, текущего синтаксиса или же традиционные делегаты — полностью локальны по отношению к коду на языке IL в точности как любой другой метод C#.

В противоположность этому интерпретируемые запросы являются *дескриптивными*. Они действуют на последовательностях, реализующих интерфейс `IQueryable<T>`, и преобразуются в операции запросов из класса `Queryable`, которые выпускают *деревья выражений*, интерпретируемые во время выполнения. Такие деревья выражений можно транслировать, скажем, в SQL-запросы, позволяя использовать LINQ для запрашивания базы данных.



Операции запросов в классе `Enumerable` могут в действительности работать с последовательностями `IQueryable<T>`. Сложность в том, что результирующие запросы всегда выполняются локально на стороне клиента — именно потому в классе `Queryable` предлагается второй набор операций запросов.

Для написания интерпретируемых запросов вам необходимо выбрать API-интерфейс, предоставляющий последовательности типа `IQueryable<T>`. Примером может служить инфраструктура *Entity Framework Core* (EF Core) от Microsoft, которая позволяет запрашивать разнообразные базы данных, включая SQL Server, Oracle, MySQL, PostgreSQL и SQLite.

Можно также сгенерировать оболочку `IQueryable<T>` для обычной перечислимой коллекции, вызвав метод `AsQueryable`. Мы опишем метод `AsQueryable` в разделе “Построение выражений запросов” далее в главе.



Интерфейс `IQueryable<T>` является расширением интерфейса `IEnumerable<T>` с дополнительными методами, предназначенными для конструирования деревьев выражений. Большую часть времени вы будете игнорировать детали, связанные с этими методами; они косвенно вызываются инфраструктурой. Интерфейс `IQueryable<T>` более подробно рассматривается в разделе “Построение выражений запросов” далее в главе.

В целях иллюстрации давайте создадим в SQL Server простую таблицу заказчиков (`Customer`) и наполним ее несколькими именами с применением следующего SQL-сценария:

```
create table Customer
(
    ID int not null primary key,
    Name varchar(30)
)
insert Customer values (1, 'Tom')
insert Customer values (2, 'Dick')
insert Customer values (3, 'Harry')
insert Customer values (4, 'Mary')
insert Customer values (5, 'Jay')
```

Располагая такой таблицей, мы можем написать на C# интерпретируемый запрос LINQ, который использует EF Core для извлечения заказчиков с именами, содержащими букву “a”:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using var dbContext = new NutshellContext();

IQueryable<string> query = from c in dbContext.Customers
    where c.Name.Contains ("a")
    orderby c.Name.Length
    select c.Name.ToUpper();

foreach (string name in query) Console.WriteLine (name);

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

// Мы более подробно обсудим класс NutshellContext в следующем разделе
public class NutshellContext : DbContext
{
    public virtual DbSet<Customer> Customers { get; set; }
    protected override void OnConfiguring (DbContextOptionsBuilder builder)
        => builder.UseSqlServer ("... строка подключения...");
```

```
protected override void OnModelCreating (ModelBuilder modelBuilder)
=> modelBuilder.Entity<Customer>().ToTable ("Customer")
    .HasKey (c => c.ID);
}
```

Инфраструктура EF Core транслирует такой запрос в следующий SQL-оператор:

```
SELECT UPPER([c].[Name])
FROM [Customers] AS [c]
WHERE CHARINDEX(N'a', [c].[Name]) > 0
ORDER BY CAST(LEN([c].[Name]) AS int)
```

Конечный результат выглядит так:

```
JAY
MARY
HARRY
```

Каким образом работают интерпретируемые запросы

Давайте выясним, как обрабатывается показанный ранее запрос.

Сначала компилятор преобразует синтаксис запросов в текущий синтаксис, поступая в точности так, как с локальными запросами:

```
IQueryable<string> query = dbContext.customers
    .Where (n => n.Name.Contains ("a"))
    .OrderBy (n => n.Name.Length)
    .Select (n => n.Name.ToUpper());
```

Далее компилятор распознает методы операций запросов. Здесь локальные и интерпретируемые запросы отличаются — интерпретируемые запросы преобразуются в операции запросов из класса `Queryable`, а не `Enumerable`.

Чтобы понять причину, необходимо взглянуть на переменную `dbContext.Customers`, являющуюся источником, на котором строится весь запрос. Переменная `dbContext.Customers` имеет тип `DbSet<T>`, реализующий интерфейс `IQueryable<T>` (подтип `IEnumerable<T>`). Таким образом, у компилятора есть выбор при распознавании `Where`: он может вызвать расширяющий метод в `Enumerable` или следующий расширяющий метод в `Queryable`:

```
public static IQueryable<TSource> Where<TSource> (this
    IQueryable<TSource> source, Expression <Func<TSource, bool>> predicate)
```

Компилятор выбирает метод `Queryable.Where`, потому что его сигнатура обеспечивает более специфичное соответствие.

Метод `Queryable.Where` принимает предикат, помещенный в оболочку типа `Expression<TDelegate>`. Тем самым компилятору сообщается о необходимости транслировать переданное лямбда-выражение, т.е. `n=>n.Name.Contains("a")`, в дерево выражения, а не в компилируемый делегат. Дерево выражения — это объектная модель, основанная на типах из пространства имен `System.Linq.Expressions`, которая может инспектироваться во время выполнения (так что инфраструктура EF Core может позже транслировать ее в SQL-оператор).

Поскольку метод `Queryable.Where` также возвращает экземпляр реализации `IQueryable<T>`, для операций `OrderBy` и `Select` происходит аналогичный процесс. Конечный результат показан на рис. 8.9. В затененном прямоугольнике представлено дерево выражения, описывающее полный запрос, которое можно обходить во время выполнения.

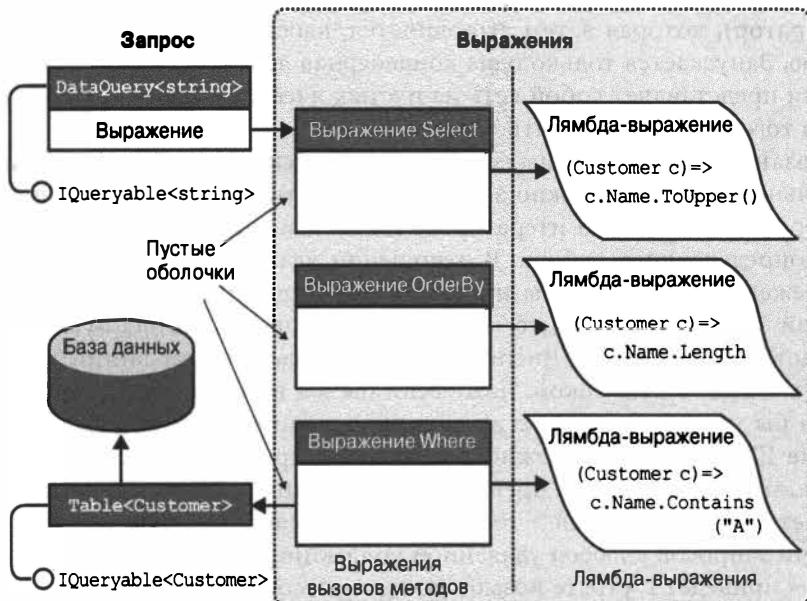


Рис. 8.9. Композиция интерпретируемого запроса

Выполнение

Интерпретируемые запросы следуют модели отложенного выполнения — подобно локальным запросам. Это означает, что SQL-оператор не генерируется вплоть до начала перечисления запроса. Кроме того, двукратное перечисление одного и того же запроса приводит к двум отправкам запроса в базу данных.

“За кулисами” интерпретируемые запросы отличаются от локальных запросов тем, каким образом они выполняются. При перечислении интерпретируемого запроса самая внешняя последовательность запускает программу, которая обходит все дерево выражения, обрабатывая его как единое целое. В нашем примере инфраструктура EF Core транслирует дерево выражения в SQL-оператор, который затем выполняется, выдавая результаты в виде последовательности.



Для работы инфраструктура EF Core должна понять схему базы данных, для чего она задействует соглашения, атрибуты кода и текущий API-интерфейс конфигурирования. Мы подробно обсудим все это позже в главе.

Ранее уже упоминалось, что запрос LINQ подобен производственной линии. Однако выполнение перечисления конвейерной ленты `IQueryable` не приводит к запуску всей производственной линии, как в случае локального запроса.

са. Взамен запускается только конвейерная лента `IQueryable` со специальным перечислителем, который вызывается на диспетчере производственной линии. Диспетчер просматривает целую конвейерную ленту, которая состоит не из скомпилированного кода, а из заглушек (выражений вызовов методов) с инструкциями, вставленными в их начало (деревья выражений). Затем диспетчер обходит все выражения, в данном случае переписывая их в единую сущность (SQL-оператор), которая затем выполняется, выдавая результаты обратно потребителю. Запускается только одна конвейерная лента; остаток производственной линии представляет собой сеть из пустых ячеек, существующих только для описания того, что должно быть сделано.

Из сказанного вытекает несколько практических последствий. Например, для локальных запросов можно записывать собственные методы запросов (довольно просто с помощью итераторов) и затем использовать их для дополнения предопределенного набора. В отношении удаленных запросов это трудно и даже нежелательно. Если вы написали расширяющий метод `MyWhere`, принимающий `IQueryable<T>`, то хотели бы помещать собственную заглушку в производственную линию. Диспетчеру производственной линии неизвестно, что делать с вашей заглушкой. Даже если бы вы вмешались в данную фазу, то получили бы решение, которое жестко привязано к конкретному поставщику наподобие EF Core без возможности работы с другими реализациями интерфейса `IQueryable`. Одно из преимуществ наличия в `Queryable` стандартного набора методов заключается в том, что они определяют *стандартный словарь* для выдачи запросов к *любой* удаленной коллекции. Попытка расширения такого словаря приведет к утрате возможности взаимодействия.

Из модели вытекает еще одно следствие: поставщик `IQueryable` может оказаться не в состоянии справиться с некоторыми запросами — даже если вы придерживаетесь стандартных методов. Инфраструктура EF Core ограничена возможностями сервера базы данных; для некоторых запросов LINQ не предусмотрена трансляция в SQL. Если вы знакомы с языком SQL, тогда имеете об этом хорошее представление, хотя иногда приходится экспериментировать, чтобы посмотреть, что именно вызвало ошибку во время выполнения. Вас может удивить, что некоторые средства *вообще* работают!

Комбинирование интерпретируемых и локальных запросов

Запрос может включать и интерпретируемые, и локальные операции. В типичном шаблоне применяются локальные операции *снаружи* и интерпретируемые компоненты *внутри*; другими словами, интерпретируемые запросы наполняют локальные запросы. Такой шаблон хорошо работает при запрашивании базы данных.

Например, предположим, что мы написали специальный расширяющий метод для объединения в пары строк из коллекции:

```
public static IEnumerable<string> Pair (this IEnumerable<string> source)
{
    string firstHalf = null;
```

```

foreach (string element in source)
{
    if (firstHalf == null)
        firstHalf = element;
    else
    {
        yield return firstHalf + ", " + element;
        firstHalf = null;
    }
}

```

Этот расширяющий метод можно использовать в запросе со смесью операций EF Core и локальных операций:

```

using var dbContext = new NutshellContext ();
IEnumerable<string> q = dbContext.Customers
    .Select (c => c.Name.ToUpper ())
    .OrderBy (n => n)
    .Pair()          // Локальная с этого момента
    .Select((n, i) => "Pair " + i.ToString () + " = " + n); // Отобразить пару
foreach (string element in q) Console.WriteLine (element);
// Pair 0 = DICK, HARRY
// Pair 1 = JAY, MARY

```

Поскольку `dbContext.Customers` имеет тип, реализующий интерфейс `IQueryable<T>`, операция `Select` преобразуется в вызов метода `Queryable.Select`. Данный метод возвращает выходную последовательность, также имеющую тип `IQueryable<T>`, поэтому операция `OrderBy` аналогично преобразуется в вызов метода `Queryable.OrderBy`. Но следующая операция запроса, `Pair`, не имеет перегруженной версии, которая принимала бы тип `IQueryable<T>` — есть только версия, принимающая менее специфичный тип `IEnumerable<T>`. Таким образом, она преобразуется в наш локальный метод `Pair` с упаковкой интерпретируемого запроса в локальный запрос. Метод `Pair` также возвращает реализацию интерфейса `IEnumerable`, а потому последующая операция `Select` преобразуется в еще одну локальную операцию. На стороне EF Core результирующий SQL-оператор эквивалентен такому коду:

```
SELECT UPPER([c].[Name]) FROM [Customers] AS [c] ORDER BY UPPER([c].[Name])
```

Оставшаяся часть работы выполняется локально. Фактически мы получаем локальный запрос (снаружи), источником которого является интерпретируемый запрос (внутри).

Метод `AsEnumerable`

Метод `Enumerable.AsEnumerable` — простейшая из всех операций запросов. Вот полное определение метода:

```

public static IEnumerable<TSource> AsEnumerable<TSource>
    (this IEnumerable<TSource> source)
{
    return source;
}

```

Он предназначен для приведения последовательности `IQueryable<T>` к `IEnumerable<T>`, заставляя последующие операции запросов привязываться к операциям из класса `Enumerable`, а не к операциям из класса `Queryable`. Это приводит к тому, что остаток запроса выполняется локально.

В целях иллюстрации предположим, что в базе данных SQL Server имеется таблица со статьями по медицине `MedicalArticles`, и с помощью EF Core необходимо извлечь все статьи, посвященные гриппу (`influenza`), резюме которых содержит менее 100 слов. Для последнего предиката потребуется регулярное выражение:

```
Regex wordCounter = new Regex(@"\b(\w|[-])+\\b");
using var dbContext = new NutshellContext ();
var query = dbContext.MedicalArticles
    .Where (article => article.Topic == "influenza" &&
        wordCounter.Matches (article.Abstract).Count < 100);
```

Проблема в том, что база данных SQL Server не поддерживает регулярные выражения, поэтому инфраструктура EF Core будет генерировать исключение, сообщая о невозможности трансляции запроса в SQL. Мы можем решить проблему за два шага: сначала извлечь все статьи, посвященные гриппу, посредством запроса EF Core, а затем локально отфильтровать сопровождающие статьи резюме, которые содержат менее 100 слов:

```
Regex wordCounter = new Regex(@"\b(\w|[-])+\\b");
using var dbContext = new NutshellContext ();
IQueryable<MedicalArticle> efQuery = dbContext.MedicalArticles
    .Where (article => article.Topic == "influenza");
IEnumerable<MedicalArticle> localQuery = efQuery
    .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Так как `efQuery` имеет тип `IEnumerable<MedicalArticle>`, второй запрос привязывается к локальным операциям запросов, обеспечивая выполнение части фильтрации на стороне клиента.

С помощью `AsEnumerable` то же самое можно сделать в единственном запросе:

```
Regex wordCounter = new Regex(@"\b(\w|[-])+\\b");
using var dbContext = new NutshellContext ();
var query = dbContext.MedicalArticles
    .Where (article => article.Topic == "influenza")
    .AsEnumerable()
    .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Альтернативой вызову `AsEnumerable` является вызов метода `ToArray` или `ToList`. Преимущество операции `AsEnumerable` в том, что она не приводит к немедленному выполнению запроса и не создает какой-либо структуры для хранения.



Перенос обработки запросов из сервера базы данных на сторону клиента может нанести ущерб производительности, особенно если обработка предусматривает извлечение дополнительных строк. Более эффективный (хотя и более сложный) способ решения предполагает применение интеграции CLR с SQL для открытия доступа к функции в базе данных, которая реализует регулярное выражение.

В главе 10 мы приведем дополнительные примеры использования комбинированных интерпретируемых и локальных запросов.

Инфраструктура EF Core

Для демонстрации интерпретируемых запросов в текущей главе и в главе 9 применяется инфраструктура EF Core. Давайте исследуем ее ключевые особенности.

Сущностные классы EF Core

Инфраструктура EF Core позволяет использовать для представления данных любой класс при условии, что он содержит открытые свойства для всех столбцов, которые планируется запрашивать.

Например, мы могли бы определить следующий сущностный класс для за-прашивания и обновления таблицы Customers в базе данных:

```
public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
```

Объект DbContext

После определения сущностных классов необходимо определить подкласс класса DbContext. Экземпляр этого класса представляет ваши сеансы, работающие с базой данных. Обычно подкласс DbContext будет содержать по одному свойству DbSet<T> для каждой сущности в модели:

```
public class NutshellContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    ...свойства для остальных таблиц...
}
```

Вот за что отвечает объект DbContext.

- Он действует как фабрика для генерирования объектов DbSet<T>, которые вы можете запрашивать.
- Он отслеживает любые изменения, которые вы вносите в свои сущности, чтобы их можно было записать обратно.
- Он предоставляет виртуальные методы, которые вы можете переопределять для конфигурирования подключения и модели.

Конфигурирование подключения

За счет переопределения метода `OnConfiguring` вы можете указывать поставщик базы данных и строку подключения:

```
public class NutshellContext : DbContext
{
    ...
    protected override void OnConfiguring (DbContextOptionsBuilder
        optionsBuilder) =>
        optionsBuilder.UseSqlServer
            ("Server=(local);Database=Nutshell;Trusted_Connection=True");
}
```

В этом примере строка подключения указывается в виде строкового литерала. Производственные приложения обычно будут извлекать ее из конфигурационного файла вроде `appsettings.json`.

`UseSqlServer` — расширяющий метод, определенный в сборке, которая входит в состав NuGet-пакета `Microsoft.EntityFrameworkCore.SqlServer`. Доступны пакеты и для других поставщиков баз данных, включая Oracle, MySQL, PostgreSQL и SQLite.



Если вы работаете с платформой ASP.NET, то можете разрешить ее инфраструктуре внедрения зависимостей предварительно конфигурировать `optionsBuilder`; в большинстве случаев такой прием позволяет вообще избежать переопределения метода `OnConfiguring`. Для этого определите конструктор в `DbContext` следующим образом:

```
public NutshellContext (DbContextOptions<NutshellContext>
    options)
    : base(options) { }
```

Если вы решили переопределить метод `OnConfiguring` (возможно, чтобы предоставить конфигурацию, если ваш объект `DbContext` используется в другом сценарии), тогда вот как можно проверить, сконфигурированы ли параметры:

```
protected override void OnConfiguring (
    DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        ...
    }
}
```

В методе `OnConfiguring` можно включать и другие параметры, в том числе ленивую загрузку (см. раздел “Ленивая загрузка” далее в главе).

Конфигурирование модели

По умолчанию инфраструктура EF Core основана на соглашениях, т.е. она выводит схему базы данных из имен вашего класса и свойств.

Вы можете переопределить стандартные соглашения с применением *текущего API-интерфейса*, переопределяя метод `OnModelCreating` и вызывая расши-

ряющие методы на параметре `ModelBuilder`. Например, можно явно указать имя таблицы базы данных для сущности `Customer`:

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
    modelBuilder.Entity<Customer>()
        .ToTable ("Customer"); // Таблица называется Customer
```

Без такого кода инфраструктура EF Core отобразила бы эту сущность на таблицу по имени “`Customers`”, а не “`Customer`” из-за наличия в `DbContext` свойства типа `DbSet<Customer>`, которое имеет имя `Customers`:

```
public DbSet<Customer> Customers { get; set; }
```



Следующий код отображает все сущности на имена таблиц, которые соответствуют именам классов сущностей (обычно имеющим форму единственного числа), а не именам свойств типа `DbSet<T>` (обычно имеющим форму множественного числа):

```
protected override void OnModelCreating (ModelBuilder modelBuilder)
{
    foreach (IMutableEntityType entityType in
        modelBuilder.Model.GetEntityTypes())
    {
        modelBuilder.Entity (entityType.Name)
            .ToTable (entityType.ClrType.Name);
    }
}
```

Текущий API-интерфейс предлагает расширенный синтаксис для конфигурирования столбцов. В показанном ниже примере мы применяем два популярных метода:

- `HasColumnName`, который отображает свойство на по-другому именованный столбец;
- `IsRequired`, который указывает на то, что столбец не допускает значения `null`.

```
protected override void OnModelCreating (ModelBuilder modelBuilder) =>
    modelBuilder.Entity<Customer> (entity =>
    {
        entity.ToTable ("Customer");
        entity.Property (e => e.Name)
            .HasColumnName ("Full Name") // Имя столбца - Full Name
            .IsRequired(); // Столбце не допускает значения null
    });

```

В табл. 8.1 перечислены наиболее важные методы в текущем API-интерфейсе.



Вместо использования текущего API-интерфейса вы можете конфигурировать свою модель путем применения специальных атрибутов к сущностным классам и свойствам (“аннотаций данных”). Такой подход менее гибок в том, что конфигурация должна устанавливаться на этапе компиляции, и менее мощный в том, что некоторые параметры можно конфигурировать только через текущий API-интерфейс.

Таблица 8.1. Методы для конфигурирования модели в текущем API-интерфейсе

Метод	Назначение	Пример
ToTable	Указывает имя таблицы базы данных	builder .Entity<Customer>() .ToTable("Customer");
HasColumnName	Указывает имя столбца для заданного свойства	builder.Entity<Customer>() .Property(c => c.Name) .HasColumnName("Full Name");
HasKey	Указывает ключ (обычно это отклонение от соглашения)	builder.Entity<Customer>() .HasKey(c => c.CustomerNr);
IsRequired	Указывает на то, что свойство требует значения (не допускает null)	builder.Entity<Customer>() .Property(c => c.Name) .IsRequired();
HasMaxLength	Указывает максимальную длину типа с переменной длинной (обычно строки), чья ширина может варьироваться	builder.Entity<Customer>() .Property(c => c.Name) HasMaxLength(60);
HasColumnType	Указывает тип данных в базе данных для столбца	builder.Entity<Purchase>() .Property(p => p.Description) HasColumnType("varchar(80)");
Ignore	Игнорирует тип	builder.Ignore<Products>();
Ignore	Игнорирует свойство типа	builder.Entity<Customer>() .Ignore(c => c.ChatName);
HasIndex	Указывает свойство (или комбинацию свойств), которое должно служить индексом в базе данных	// Составной индекс: // builder.Entity<Purchase>() // .HasIndex(p => // new { p.Date, p.Price }); // Уникальный индекс // на одном свойстве: builder .Entity<MedicalArticle>() .HasIndex(a => a.Topic) .IsUnique();
HasOne	См. раздел “Навигационные свойства” далее в главе	builder.Entity<Purchase>() .HasOne(p => p.Customer) .WithMany(c => c.Purchases);
hasMany	См. раздел “Навигационные свойства” далее в главе	builder.Entity<Customer>() .HasMany(c => c.Purchases) .WithOne(p => p.Customer);

Создание базы данных

Инфраструктура EF Core поддерживает подход “сначала код”, т.е. вы можете начать с определения сущностных классов и затем предложить EF Core создать базу данных. Самый простой способ создания базы данных предусматривает вызов следующего метода на экземпляре DbContext:

```
dbContext.Database.EnsureCreated();
```

Однако более удачный подход предполагает использование функционального средства *миграций* EF Core, которое не только создает базу данных, но и конфигурирует ее так, что инфраструктура EF Core может автоматически обновлять схему в будущем, когда ваши сущностные классы изменятся. Находясь в консоли диспетчера пакетов Visual Studio, вы можете включить миграции и потребовать создание базы данных с помощью следующих команд:

```
Install-Package Microsoft.EntityFrameworkCore.Tools  
Add-Migration InitialCreate  
Update-Database
```

Первая команда устанавливает инструменты для управления инфраструктурой EF Core из среды Visual Studio. Вторая команда генерирует специальный класс C#, известный как кодовая миграция, который содержит инструкции для создания базы данных. Последняя команда запускает эти инструкции относительно строки подключения к базе данных, указанной в конфигурационном файле проекта приложения.

Использование `DbContext`

После определения сущностных классов и подкласса `DbContext` вы можете создать экземпляр `DbContext` и запрашивать базу данных:

```
using var dbContext = new NutshellContext();  
Console.WriteLine (dbContext.Customers.Count());  
// Выполняется SELECT COUNT(*) FROM [Customer] AS [c]
```

Экземпляр `DbContext` можно применять также для записи в базу данных. Следующий код вставляет строку в таблицу `Customers`:

```
using var dbContext = new NutshellContext();  
Customer cust = new Customer()  
{  
    Name = "Sara Wells"  
};  
dbContext.Customers.Add (cust);  
dbContext.SaveChanges(); // Записывает изменения обратно в базу данных
```

Показанный далее код запрашивает у базы данных только что вставленную строку с информацией о заказчике:

```
using var dbContext = new NutshellContext();  
Customer cust = dbContext.Customers  
.Single (c => c.Name == "Sara Wells")
```

Приведенный ниже код обновляет имя этого заказчика и записывает изменение в базу данных:

```
cust.Name = "Dr. Sara Wells";  
dbContext.SaveChanges();
```



Операция `Single` идеально подходит для извлечения строки по первичному ключу. В отличие от `First` в случае возвращения более одного элемента она генерирует исключение.

Отслеживание объектов

Экземпляр `DbContext` отслеживает все сущности, экземпляры которых он создает, так что при запросе тех же самых строк в таблице он может выдавать те же самые сущности. Другими словами, за время своего существования контекст никогда не выпустит две отдельных сущности, которые относятся к одной и той же строке в таблице (когда строка идентифицируется первичным ключом). Такая возможность называется *отслеживанием объектов*.

В целях иллюстрации предположим, что заказчик, имя которого является первым в алфавитном порядке, также имеет наименьший идентификатор. В следующем примере `a` и `b` будут ссылаться на один и тот же объект:

```
using var dbContext = new NutshellContext ();
Customer a = dbContext.Customers.OrderBy (c => c.Name).First();
Customer b = dbContext.Customers.OrderBy (c => c.ID).First();
```

Освобождение экземпляров `DbContext`

Несмотря на то что класс `DbContext` реализует интерфейс `IDisposable`, можно (в общем случае) обойтись без освобождения его экземпляров. Освобождение принудительно закрывает подключение контекста, но обычно поступать подобным образом вовсе не обязательно, т.к. инфраструктура EF Core закрывает подключения автоматически всякий раз, когда завершается извлечение результатов из запроса.

Преждевременное освобождение контекста может в действительности окаться проблематичным из-за ленивого вычисления. Взгляните на следующий код:

```
IQueryable<Customer> GetCustomers (string prefix)
{
    using (var dbContext = new NutshellContext ())
        return dbContext.Customers
            .Where (c => c.Name.StartsWith (prefix));
}
...
foreach (Customer c in GetCustomers ("a"))
    Console.WriteLine (c.Name);
```

Такой код потерпит неудачу, потому что запрос вычисляется при его перечислении, которое происходит *после освобождения* связанного с ним экземпляра `DbContext`.

Тем не менее, ниже приведены некоторые предостережения, о которых следует помнить в случае, если контексты не освобождаются.

- Как правило, объект подключения должен освобождать все управляемые ресурсы в методе `Close`. В то время как это справедливо для класса `SqlConnection`, сторонние объекты подключений теоретически могут удерживать ресурсы открытыми, если метод `Close` вызывался, а метод `Dispose` — нет (хотя такой подход, вероятно, нарушит контракт, определенный методом `IDbConnection.Close`).

- Если вы вручную вызываете метод `GetEnumerator` на запросе (вместо применения оператора `foreach`) и затем забываете либо освободить перечислитель, либо воспользоваться последовательностью, то подключение останется открытым. Освобождение экземпляра `DbContext` предоставляет страховку для таких сценариев.
- Некоторые разработчики считают гораздо более аккуратным подходом освобождение контекстов (и всех объектов, которые реализуют интерфейс `IDisposable`).

Чтобы освободить контексты явно, потребуется передать экземпляр `DbContext` в методы вроде `GetCustomers` и избежать описанных выше проблем. В сценариях вроде ASP.NET Core MVC, где экземпляр контекста предоставляется через внедрение зависимостей (dependency injection — DI), временем жизни контекста будет управлять инфраструктура DI. Он будет создаваться, когда единица работы (такая как обработка HTTP-запроса в контроллере) начинается, и освобождаться, когда единица работы заканчивается.

Давайте посмотрим, что происходит, когда инфраструктура EF Core сталкивается со вторым запросом. Она начинает с отправки запроса базе данных и в ответ получает одиночную строку. Затем инфраструктура читает первичный ключ полученной строки и производит его поиск в кеше сущностей контекста. Обнаружив совпадение, она возвращает существующий объект **без обновления любых значений**. Таким образом, если другой пользователь только что обновил имя текущего заказчика в базе данных, то новое значение будет проигнорировано. Это важно для устранения нежелательных побочных эффектов (объект `Customer` может использоваться где-то в другом месте) и также для управления параллелизмом. Если вы изменили свойства объекта `Customer` и пока еще не вызвали метод `SaveChanges`, то определенно не хотите, чтобы данные были автоматически перезаписаны.



Отключить отслеживание объектов можно, присоединив к запросу расширяющий метод `AsNoTracking` или установив свойство `ChangeTracker.QueryTrackingBehavior` объекта контекста в `QueryTrackingBehavior.NoTracking`. Неотслеживаемые запросы удобны, когда данные используются только для чтения, т.к. они улучшают производительность и снижают потребление памяти.

Чтобы получить актуальную информацию из базы данных, потребуется либо создать новый экземпляр контекста, либо вызвать его метод `Reload`:

```
dbContext.Entry(myCustomer).Reload();
```

Рекомендуется применять свежий экземпляр `DbContext` на единицу работы, чтобы необходимость в ручной перезагрузке сущности возникала редко.

Отслеживание изменений

Когда вы изменяете значение свойства в сущности, загруженной через `DbContext`, инфраструктура EF Core распознает изменение и соответствующим

образом обновляет базу данных при вызове SaveChanges. Для этого она создает моментальный снимок состояния сущностей, загруженных посредством вашего подкласса DbContext, и сравнивает текущее состояние с первоначальным состоянием во время вызова метода SaveChanges (или, как вскоре будет показано, при ручном отслеживании изменений запросов). Вот как вы можете выполнить перечисление отслеженных изменений в DbContext:

```
foreach (var e in dbContext.ChangeTracker.Entries())
{
    Console.WriteLine($"{e.Entity.GetType().FullName} is {e.State}");
    foreach (var m in e.Members)
        Console.WriteLine(
            $" {m.Metadata.Name}: '{m.CurrentValue}' modified: {m.IsModified}");
}
```

В случае вызова метода SaveChanges инфраструктура EF Core использует информацию в ChangeTracker при построении SQL-операторов, которые будут обновлять базу данных для соответствия изменениям, внесенным в ваши объекты. Она выпускает операторы вставки для добавления новых строк, операторы обновления для модификации данных и операторы удаления для исключения строк, удаленных из объектного графа в подклассе DbContext. Принимаются во внимание любые экземпляры TransactionScope; если они отсутствуют, тогда все операторы помещаются в новую транзакцию.

Вы можете оптимизировать отслеживание изменений, реализовав в своих сущностных классах интерфейс INotifyPropertyChanged и дополнительно интерфейс INotifyPropertyChanging. Первый позволяет инфраструктуре EF Core избегать накладных расходов на сравнение модифицированных и первоначальных сущностей, а второй позволяет EF Core вообще не хранить первоначальные значения. После реализации упомянутых интерфейсов для активизации оптимизированного отслеживания изменений понадобится вызвать метод HasChangeTrackingStrategy на экземпляре ModelBuilder при конфигурировании модели.

Навигационные свойства

Навигационные свойства дают возможность выполнять следующие действия:

- запрашивать связанные таблицы без необходимости в ручном соединении;
- вставлять, удалять и обновлять связанные строки, не обновляя явно внешние ключи.

Например, предположим, что у заказчика может быть несколько покупок. Мы можем представить отношение “один ко многим” между Customer и Purchase с помощью таких сущностей:

```
public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
    //Дочернее навигационное свойство, которое обязано иметь тип ICollection<T>;
    public virtual List<Purchase> Purchases {get;set;} = new List<Purchase>();
}
```

```

public class Purchase
{
    public int ID { get; set; }
    public DateTime Date { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public int CustomerID? { get; set; } // Поле внешнего ключа
    public Customer Customer { get; set; } // Родительское навигационное
                                            // свойство
}

```

На основе приведенных выше сущностей инфраструктура EF Core способна сделать вывод о том, что CustomerID является внешним ключом к таблице Customers, поскольку имя CustomerID следует популярному соглашению об именовании. Если бы мы запросили у инфраструктуры EF Core создание базы данных из таких сущностей, то она создала бы ограничение внешнего ключа между Purchase.CustomerID и Customer.ID.



Если инфраструктуре EF Core не удается вывести отношение, тогда вы можете сконфигурировать его явно в методе OnModelCreating:

```

modelBuilder.Entity<Purchase>()
    .HasOne (e => e.Customer)
    .WithMany (e => e.Purchases)
    .HasForeignKey (e => e.CustomerID);

```

После настройки этих навигационных свойств появляется возможность формулировать запросы вроде показанных ниже:

```
var customersWithPurchases = Customers.Where (c => c.Purchases.Any());
```

В главе 9 мы более подробно обсудим, как записывать запросы подобного рода.

Добавление и удаление сущностей из навигационных коллекций

Когда вы добавляете новые сущности к навигационному свойству типа коллекции, инфраструктура EF Core автоматически заполняет внешние ключи при вызове SaveChanges:

```

Customer cust = dbContext.Customers.Single (c => c.ID == 1);
Purchase p1 = new Purchase { Description="Bike", Price=500 };
Purchase p2 = new Purchase { Description="Tools", Price=100 };

cust.Purchases.Add (p1);
cust.Purchases.Add (p2);

dbContext.SaveChanges();

```

В настоящем примере инфраструктура EF Core автоматически записывает 1 в столбец CustomerID каждой новой строки с информацией о покупке и записывает идентификатор, генерируемый базой данных для каждой строки с информацией о покупке, в Purchase.ID.

Когда вы удаляете сущность из навигационного свойства типа коллекции и вызываете SaveChanges, инфраструктура EF Core либо очистит поле внешнего

ключа, либо удалит соответствующую строку из базы данных, что зависит от того, каким образом было сконфигурировано или выведено отношение. В нашем случае мы определили `Purchase.CustomerID` как целое число, допускающее `null` (с тем, чтобы можно было представлять покупки без заказчика, или денежные операции), поэтому удаление покупки у заказчика приведет к очистке ее поля внешнего ключа, а не к ее удалению из базы данных.

Загрузка навигационных свойств

Когда инфраструктура EF Core заполняет сущность, (по умолчанию) она не заполняет ее навигационные свойства:

```
using var dbContext = new NutshellContext();
var cust = dbContext.Customers.First();
Console.WriteLine (cust.Purchases.Count); // Всегда 0
```

Одно из решений предусматривает использование расширяющего метода `Include`, который инструктирует EF Core о необходимости энергичной загрузки навигационных свойств:

```
var cust = dbContext.Customers
    .Include (c => c.Purchases)
    .Where (c => c.ID == 2).First();
```

Другое решение связано с применением проецирования. Такой прием особенно удобен, если нужно работать только с некоторыми свойствами сущности, потому что он сокращает объем передаваемых данных:

```
var custInfo = dbContext.Customers
    .Where (c => c.ID == 2)
    .Select (c => new
    {
        Name = c.Name,
        Purchases = c.Purchases.Select (p => new { p.Description, p.Price })
    })
    .First();
```

Обе методики информируют инфраструктуру EF Core о том, какие данные требуются, так что они могут быть извлечены в одиночном запросе к базе данных. Также можно вручную инструктировать инфраструктуру EF Core относительно заполнения навигационного свойства по мере необходимости:

```
dbContext.Entry (cust).Collection (b => b.Purchases).Load();
// Коллекция cust.Purchases теперь заполнена.
```

Такой подход называется *явной загрузкой*. В отличие от предшествующих подходов он порождает дополнительное обращение к серверу базы данных.

Ленивая загрузка

Еще один подход к загрузке навигационных свойств называется *ленивой загрузкой*. Когда она включена, инфраструктура EF Core заполняет навигационные свойства по требованию, генерируя для каждого сущностного класса класс-посредник, который перехватывает попытки доступа к незагруженным навигационным свойствам. Чтобы это работало, каждое навигационное свойство должно

быть виртуальным, а класс, в котором оно определено, обязан быть наследуемым (не запечатанным). Кроме того, контекст не должен быть освобожден, когда происходит ленивая загрузка, чтобы можно было выполнить дополнительный запрос к базе данных.

Включить ленивую загрузку можно в методе `OnConfiguring` подкласса `DbContext`:

```
protected override void OnConfiguring (DbContextOptionsBuilder  
optionsBuilder)  
{  
    optionsBuilder  
.UseLazyLoadingProxies()  
    ...  
}
```

(Также понадобится добавить ссылку на NuGet-пакет `Microsoft.EntityFrameworkCore.Proxies`.)

За ленивую загрузку приходится расплачиваться тем, что инфраструктура EF Core обязана делать дополнительный запрос к базе данных при каждом обращении к незагруженному навигационному свойству. Если таких запросов много, тогда чрезмерные обращения к серверу могут привести к снижению производительности.



При включенной ленивой загрузке типом времени выполнения ваших классов будет класс-посредник, выведенный из сущностного класса; например:

```
using var dbContext = new NutshellContext();  
var cust = dbContext.Customers.First();  
Console.WriteLine (cust.GetType());  
// Castle.Proxies.CustomerProxy
```

Отложенное выполнение

Запросы EF Core подчиняются отложенному выполнению в точности как локальные запросы. Это позволяет строить запросы постепенно. Тем не менее, существует один аспект, в котором инфраструктура EF Core поддерживает специальную семантику отложенного выполнения, и возникает он в ситуации, когда подзапрос находится внутри выражения `Select`.

В локальных запросах получается двойное отложенное выполнение, поскольку с функциональной точки зрения осуществляется выборка последовательности запросов. Таким образом, если перечислять внешнюю результирующую последовательность, но не перечислять внутренние последовательности, то подзапрос никогда не выполнится.

В EF Core подзапрос выполняется в то же самое время, когда выполняется главный внешний запрос. В итоге появляется возможность избежать излишних обращений к серверу.

Скажем, следующий запрос выполняется в одиночном обращении при достижении первого оператора `foreach`:

```
using var dbContext = new NutshellContext ();
var query = from c in dbContext.Customers
            select
                from p in c.Purchases
                select new { c.Name, p.Price };

foreach (var customerPurchaseResults in query)
    foreach (var namePrice in customerPurchaseResults)
        Console.WriteLine($"'{namePrice.Name}' spent {namePrice.Price}");
```

Любые навигационные свойства, которые спроектированы явно, полностью заполняются в единственном обращении к серверу:

```
var query = from c in dbContext.Customers
            select new { c.Name, c.Purchases };

foreach (var row in query)
    foreach (Purchase p in row.Purchases) // Дополнительные обращения
                                            // к серверу отсутствуют
        Console.WriteLine(row.Name + " spent " + p.Price);
```

Но если проводить перечисление навигационного свойства, предварительно не спроектировав или энергично не загрузив его, то будут применяться правила отложенного выполнения. В приведенном ниже примере инфраструктура EF Core выполняет еще один запрос свойства Purchases на каждой итерации цикла (предполагается, что ленивая загрузка включена):

```
foreach (Customer c in dbContext.Customers.ToArray())
    foreach (Purchase p in c.Purchases) // Еще одно обращение к серверу
        Console.WriteLine(c.Name + " spent " + p.Price);
```

Такая модель полезна, когда нужно выполнять внутренний цикл избирательно, основываясь на проверке, которая может быть сделана только на стороне клиента:

```
foreach (Customer c in context.Customers)
    if (myWebService.HasBadCreditHistory(c.ID))
        foreach (Purchase p in c.Purchases) // Еще одно обращение к серверу
            Console.WriteLine(...);
```



Обратите внимание на использование `ToArray` в предшествующих двух запросах. По умолчанию SQL Server не может инициировать новый запрос, пока обрабатываются результаты текущего запроса. Вызов `ToArray` материализует заказчиков, чтобы можно было выпустить дополнительные запросы с целью извлечения покупок для каждого заказчика. Имеется возможность сконфигурировать SQL Server для включения множества активных результирующих наборов (`multiple active result set — MARS`), дополнив строку подключения к базе данных конструкцией `;MultipleActiveResultSets=True`. Применяйте наборы MARS с осторожностью, т.к. они способны замаскировать многословное проектное решение базы данных, которое можно было бы улучшить за счет энергичной загрузки и/или проектирования требующихся данных.

(Подзапросы Select более детально исследуются в разделе “Выполнение проектирования” главы 9.)

Построение выражений запросов

Когда возникала необходимость в динамически сформированных запросах, ранее в главе мы условно соединяли в цепочки операции запросов. Хотя такой прием вполне адекватен для многих сценариев, иногда требуется работать на более детализированном уровне и динамически составлять лямбда-выражения, которые передаются операциям.

В настоящем разделе мы предполагаем, что класс Product определен так:

```
public class Product
{
    public int ID { get; set; }
    public string Description { get; set; }
    public bool Discontinued { get; set; }
    public DateTime LastSale { get; set; }
}
```

Сравнение делегатов и деревьев выражений

Вспомните, что:

- локальные запросы, которые используют операции Enumerable, принимают делегаты;
- интерпретируемые запросы, которые используют операции Queryable, принимают деревья выражений.

В этом легко удостовериться, сравнив сигнатуры операции Where в классах Enumerable и Queryable:

```
public static IEnumerable<TSource> Where<TSource> (this
    I Enumerable<TSource> source, Func<TSource, bool> predicate)
public static IQueryable<TSource> Where<TSource> (this
    IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate)
```

Лямбда-выражение, внедренное в запрос, выглядит идентично вне зависимости от того, привязано оно к операциям класса Enumerable или к операциям класса Queryable:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product> q2 = sqlProducts.Where (p => !p.Discontinued);
```

Однако в случае присваивания лямбда-выражения промежуточной переменной потребуется принять явное решение относительно ее преобразования в делегат (т.е. Func<>) или в дерево выражения (т.е. Expression<Func<>>). В следующем примере переменные predicate1 и predicate2 не являются взаимозаменяемыми:

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where (predicate1);

Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where (predicate2);
```

Компиляция деревьев выражений

Дерево выражения можно преобразовать в делегат, вызвав метод `Compile`. Прием особенно полезен при написании методов, которые возвращают многократно применяемые выражения. В целях иллюстрации давайте добавим в класс `Product` статический метод, возвращающий предикат, который вычисляется в `true`, если товар не снят с производства и был продан на протяжении последних 30 дней:

```
public class Product
{
    public static Expression<Func<Product, bool>> IsSelling()
    {
        return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays (-30);
    }
}
```

Только что написанный метод можно использовать как в интерпретируемых, так и в локальных запросах:

```
void Test()
{
    var dbContext = new NutshellContext();
    Product[] localProducts = dbContext.Products.ToArray();

    IQueryable<Product> sqlQuery = dbContext.Products.Where(Product.IsSelling());
    IEnumerable<Product> localQuery =
        localProducts.Where (Product.IsSelling().Compile());
}
```



.NET не предоставляет API-интерфейса для преобразования в обратном направлении, т.е. из делегата в дерево выражения, что делает деревья выражений более универсальными.

Метод `AsQueryable`

Операция `AsQueryable` позволяет записывать целые запросы, которые могут запускаться в отношении локальных или удаленных последовательностей:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
    return from p in input
           where ...
           order by ...
           select p;
}

void Test()
{
    var dbContext = new NutshellContext();
    Product[] localProducts = dbContext.Products.ToArray();
    var sqlQuery = FilterSortProducts (dbContext.Products);
    var localQuery = FilterSortProducts (localProducts.AsQueryable());
    ...
}
```

Операция `AsQueryable` помещает локальную последовательность в оболочку `IQueryable<T>`, так что последующие операции запросов преобразуются в деревья выражений. Если позже выполнить перечисление результата, тогда деревья выражений неявно скомпилируются (за счет небольшого снижения производительности), и локальная последовательность будет перечисляться обычным образом.

Деревья выражений

Ранее уже упоминалось, что неявное преобразование из лямбда-выражения в класс `Expression<TDelegate>` заставляет компилятор C# генерировать код, который строит дерево выражения. Приложив небольшие усилия по программированию, то же самое можно сделать вручную во время выполнения — другими словами, динамически построить дерево выражения с нуля. Результат можно привести к типу `Expression<TDelegate>` и применять в запросах EF Core или скомпилировать в обычновенный делегат, вызвав метод `Compile`.

DOM-модель выражения

Дерево выражения — это миниатюрная кодовая DOM-модель. Каждый узел в дереве представлен типом из пространства имен `System.Linq.Expressions`, которые показаны на рис. 8.10.

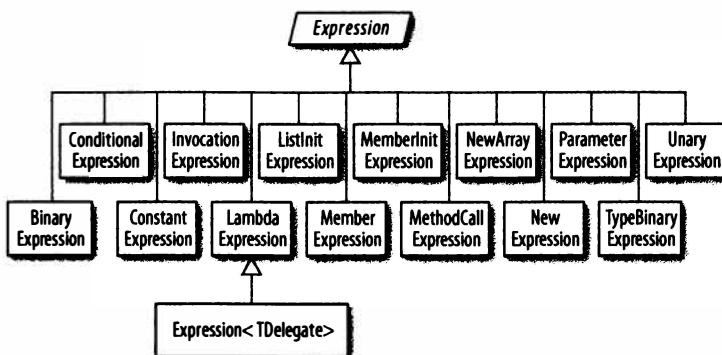


Рис. 8.10. Типы выражений

Базовым классом для всех узлов является (необщечастный) класс `Expression`. Обобщенный класс `Expression<TDelegate>` в действительности означает “типовизированное лямбда-выражение” и мог бы называться `LambdaExpression<TDelegate>`, если бы следующая запись не выглядела настолько неуклюжей:

```
LambdaExpression<Func<Customer, bool>> f = ...
```

Базовым типом `Expression<T>` является (необщечастный) класс `LambdaExpression`. Класс `LambdaExpression` обеспечивает унификацию типов для деревьев лямбда-выражений: любой типизированный экземпляр `Expression<T>` может быть приведен к типу `LambdaExpression`.

Отличие `LambdaExpression` от обычного класса `Expression` связано с тем, что лямбда-выражения имеют *параметры*.

Чтобы создать дерево выражения, не нужно создавать экземпляры типов узлов напрямую; взамен следует вызывать статические методы, предлагаемые классом `Expression`, такие как `Add`, `And`, `Call`, `Constant`, `LessThan` и т.д.

На рис. 8.11 представлено дерево выражения, которое создается в результате следующего присваивания:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

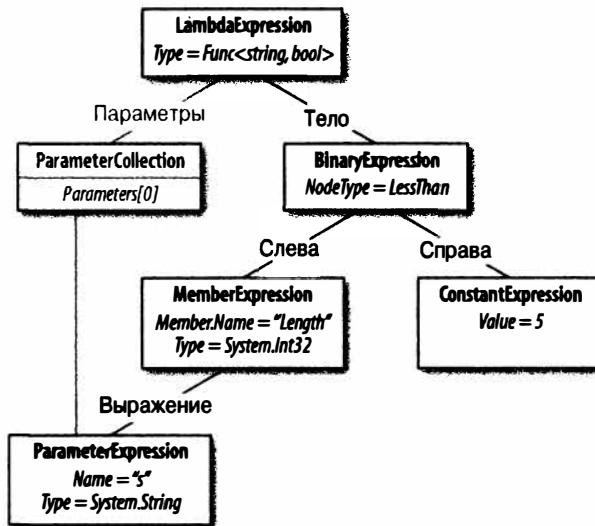


Рис. 8.11. Дерево выражения

Это можно продемонстрировать с помощью такого кода:

```
Console.WriteLine(f.Body.NodeType); // LessThan
Console.WriteLine(((BinaryExpression) f.Body).Right); // 5
```

Давайте теперь построим такое выражение с нуля. Принцип заключается в том, чтобы начать с нижней части дерева и работать, двигаясь вверх. В самой нижней части дерева находится экземпляр класса `ParameterExpression` — параметр лямбда-выражения по имени `s` типа `string`:

```
ParameterExpression p = Expression.Parameter (typeof (string), "s");
```

На следующем шаге строятся экземпляры классов `MemberExpression` и `ConstantExpression`. В первом случае необходимо получить доступ к свойству `Length` параметра `s`:

```
MemberExpression stringLength = Expression.Property (p, "Length");
ConstantExpression five = Expression.Constant (5);
```

Далее создается сравнение `LessThan` (меньше чем):

```
BinaryExpression comparison = Expression.LessThan (stringLength, five);
```

На финальном шаге конструируется лямбда-выражение, которое связывает свойство `Body` выражения с коллекцией параметров:

```
Expression<Func<string, bool>> lambda  
= Expression.Lambda<Func<string, bool>> (comparison, p);
```

Удобный способ тестирования построенного лямбда-выражения предусматривает его компиляцию в делегат:

```
Func<string, bool> runnable = lambda.Compile();  
Console.WriteLine (runnable ("kangaroo"));           // False  
Console.WriteLine (runnable ("dog"));                // True
```



Выяснить тип выражения, который должен использоваться, проще всего, просмотрев существующее лямбда-выражение в отладчике Visual Studio.

Дополнительные рассуждения по данной теме можно найти по адресу <http://www.albahari.com/expressions/>.



Операции LINQ

В настоящей главе подробно описаны все операции запросов LINQ. Для справочных целей в разделах “Выполнение проектирования” и “Выполнение соединения” далее в главе раскрывается несколько концептуально важных областей:

- проектирование иерархий объектов;
- соединение с помощью Select, SelectMany, Join и GroupJoin;
- выражения запросов с множеством переменных диапазона.

Во всех примерах главы предполагается, что массив names определен следующим образом:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

В примерах, где запрашивается база данных, переменная dbContext создается так, как показано ниже:

```
var dbContext = new NutshellContext();
```

А вот определение класса NutshellContext:

```
public class NutshellContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Purchase> Purchases { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Customer>(entity =>
        {
            entity.ToTable("Customer");
            entity.Property(e => e.Name).IsRequired(); // Столбец не допускает
                                                       // значение null
        });
        modelBuilder.Entity<Purchase>(entity =>
        {
            entity.ToTable("Purchase");
            entity.Property(e => e.Date).IsRequired();
            entity.Property(e => e.Description).IsRequired();
        });
    }
}
```

```

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
    public virtual List<Purchase> Purchases { get; set; }
        = new List<Purchase>();
}
public class Purchase
{
    public int ID { get; set; }
    public int? CustomerID { get; set; }
    public DateTime Date { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public virtual Customer Customer { get; set; }
}

```



Все примеры, которые рассматриваются в главе, предварительно загружены в LINQPad вместе с примером базы данных, имеющей подходящую схему. Загрузить LINQPad можно на веб-сайте <http://www.linqpad.net>.

Ниже приведены соответствующие определения таблиц SQL Server:

```

CREATE TABLE Customer (
    ID int NOT NULL IDENTITY PRIMARY KEY,
    Name nvarchar(30) NOT NULL
)
CREATE TABLE Purchase (
    ID int NOT NULL IDENTITY PRIMARY KEY,
    CustomerID int NOT NULL REFERENCES Customer(ID),
    Date datetime NOT NULL,
    Description nvarchar(30) NOT NULL,
    Price decimal NOT NULL
)

```

Обзор

В этом разделе будет представлен обзор стандартных операций запросов, которые делятся на три категории:

- последовательность на входе, последовательность на выходе (последовательность → последовательность);
- последовательность на входе, одиничный элемент или скалярное значение на выходе (последовательность → элемент или значение);
- ничего на входе, последовательность на выходе (ничего → последовательность), т.е. методы генерации.

Сначала мы рассмотрим каждую из трех категорий и укажем, какие операции запросов она включает, а затем перейдем к детальному описанию индивидуальных операций.

Последовательность → последовательность

В данную категорию попадает большинство операций запросов — они принимают одну или более последовательностей на входе и выпускают одиночную выходную последовательность. На рис. 9.1 показаны операции, которые реорганизуют форму последовательностей.

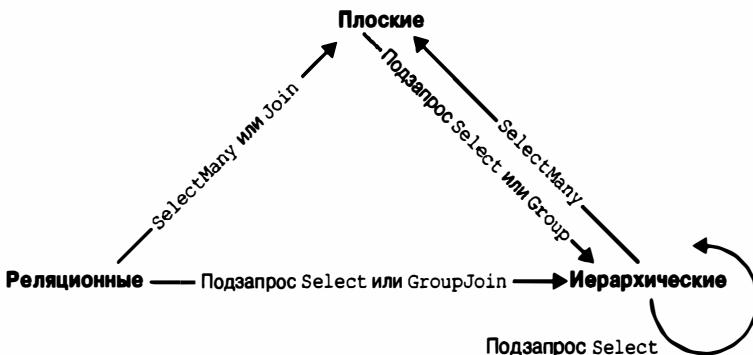


Рис. 9.1. Операции, изменяющие форму последовательностей

Фильтрация

`IEnumerable<TSource> → IEnumerable<TSource>`

Возвращает подмножество исходных элементов:

`Where, Take, TakeLast, TakeWhile, Skip, SkipLast, SkipWhile, Distinct, DistinctBy`

Проектирование

`IEnumerable<TSource> → IEnumerable<TResult>`

Трансформирует каждый элемент с помощью лямбда-функции. Операция `SelectMany` выравнивает вложенные последовательности; операции `Select` и `SelectMany` выполняют внутренние соединения, левые внешние соединения, перекрестные соединения и неэкви соединения с помощью EF Core:

`Select, SelectMany`

Соединение

`IEnumerable<TOuter>, IEnumerable<TInner> → IEnumerable<TResult>`

Объединяет элементы одной последовательности с элементами другой. Операции `Join` и `GroupJoin` спроектированы для эффективной работы с локальными запросами и поддерживают внутренние и левые внешние соединения. Операция `Zip` перечисляет две последовательности за раз, применяя функцию к каждой паре элементов. Вместо имен `TOuter` и `TInner` для аргументов типов в операции `Zip` используются имена `TFirst` и `TSecond`:

`IEnumerable<TFirst>, IEnumerable<TSecond> → IEnumerable<TResult>`
`Join, GroupJoin, Zip`

Упорядочение

`IEnumerable<TSource> → IOrderedEnumerable<TSource>`

Возвращает переупорядоченную последовательность:

`OrderBy, ThenBy, Reverse`

Группирование

`IEnumerable<TSource> → IEnumerable<IGrouping<TSource, TElement>>`

`IEnumerable<TSource> → IEnumerable<TElement[]>`

Группирует последовательность в подпоследовательности:

`GroupBy, Chunk`

Операции над множествами

`IEnumerable<TSource>, IEnumerable<TSource> → IEnumerable<TSource>`

Принимает две последовательности одного и того же типа и возвращает их общность, сумму или разницу:

`Concat, Union, UnionBy, Intersect, IntersectBy, Except, ExceptBy`

Методы преобразования: импортование

`IEnumerable → IEnumerable<TResult>`

`OfType, Cast`

Методы преобразования: экспортование

`IEnumerable<TSource> → массив, список, словарь, объект Lookup или последовательность:`

`ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable`

Последовательность → элемент или значение

Описанные ниже операции запросов принимают входную последовательность и выдают одиночный элемент или значение.

Операции над элементами

`IEnumerable<TSource> → TSource`

Выбирает одиночный элемент из последовательности:

`First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault, MinBy, MaxBy, DefaultIfEmpty`

Методы агрегирования

`IEnumerable<TSource> → скалярное значение`

Выполняет вычисление над последовательностью, возвращая скалярное значение (обычно число):

`Aggregate, Average, Count, LongCount, Sum, Max, Min`

Квантификаторы

IEnumerable<TSource> → значение bool

Агрегация, возвращающая true или false:

All, Any, Contains, SequenceEqual

Ничего → последовательность

К третьей (и последней) категории относятся операции, которые строят выходную последовательность с нуля.

Методы генерации

Ничего → IEnumerable<TResult>

Производит простую последовательность:

Empty, Range, Repeat

Выполнение фильтрации

IEnumerable<TSource> → IEnumerable<TSource>

Метод	Описание	Эквиваленты в SQL
Where	Возвращает подмножество элементов, удовлетворяющих заданному условию	WHERE
Take	Возвращает первые count элементов и отбрасывает остальные	WHERE ROW_NUMBER()... или подзапрос TOP n
Skip	Пропускает первые count элементов и возвращает остальные	WHERE ROW_NUMBER()... или NOT IN (SELECT TOP n...)
TakeLast	Возвращает только последние count элементов	Генерируется исключение
SkipLast	Пропускает последние count элементов	Генерируется исключение
TakeWhile	Выдает элементы из входной последовательности до тех пор, пока предикат не станет равным false	Генерируется исключение
SkipWhile	Пропускает элементы из входной последовательности до тех пор, пока предикат не станет равным false, после чего возвращает остальные элементы	Генерируется исключение
Distinct, DistinctBy	Возвращает последовательность, из которой исключены дубликаты	SELECT DISTINCT...



Информация в колонке “Эквиваленты в SQL” справочных таблиц в настоящей главе не обязательно соответствует тому, что будет производить реализация интерфейса `IQueryable`, такая как EF Core. Взамен в ней показано то, что обычно применяется для выполнения той же работы при написании SQL-запроса вручную. Если отсутствует простая трансляция, тогда ячейка в этой колонке остается пустой. Если же никакой трансляции вообще нет, то указывается примечание “Генерируется исключение”. Код реализации в `Enumerable`, когда он приводится, не включает проверку на предмет `null` для аргументов и предикатов индексации.

Каждый метод фильтрации всегда выдает либо такое же, либо меньшее количество элементов по сравнению с начальным их числом. Получить большее количество элементов невозможно! Кроме того, на выходе получаются идентичные элементы; они никак не трансформируются.

Where

Аргумент	Тип
Исходная последовательность	<code>IEnumerable<TSource></code>
Предикат	<code>TSource => bool</code> или <code>(TSource, int) => bool</code> *

* Запрещено в LINQ to SQL и Entity Framework.

Синтаксис запросов

`where выражение-bool`

Реализация в `Enumerable`

Если оставить в стороне проверки на равенство `null`, то внутренняя реализация операции `Enumerable.Where` функционально эквивалентна следующему коду:

```
public static IEnumerable<TSource> Where<TSource>
    (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
    foreach (TSource element in source)
        if (predicate (element))
            yield return element;
}
```

Обзор

Операция `Where` возвращает элементы входной последовательности, которые удовлетворяют заданному предикату. Например:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));
// Harry
// Mary
// Jay
```

Или в синтаксисе запросов:

```
IEnumerable<string> query = from n in names  
    where n.EndsWith ("y")  
    select n;
```

Конструкция `where` может встречаться в запросе более одного раза, перемежаясь с конструкциями `let`, `orderby` и `join`:

```
from n in names  
where n.Length > 3  
let u = n.ToUpper()  
where u.EndsWith ("Y")  
select u;  
  
// HARRY  
// MARY
```

К таким запросам применяются стандартные правила области видимости C#. Другими словами, на переменную нельзя ссылаться до ее объявления с помощью переменной диапазона или конструкции `let`.

Индексированная фильтрация

Предикат операции `Where` дополнительно принимает второй аргумент типа `int`. В него помещается позиция каждого элемента внутри входной последовательности, позволяя предикату использовать эту информацию в своем решении по фильтрации. Например, следующий запрос обеспечивает пропуск каждого второго элемента:

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 == 0);  
  
// Tom  
// Harry  
// Jay
```

Попытка применения индексированной фильтрации в EF Core приводит к генерации исключения.

Сравнения с помощью SQL-операции `LIKE` в EF Core

Следующие методы, выполняемые на типе `string`, транслируются в SQL-операцию `LIKE`:

`Contains`, `StartsWith`, `EndsWith`

Например, `c.Name.Contains ("abc")` транслируется в конструкцию `customer.Name LIKE '%abc%'` (точнее, в ее параметризованную версию).

Метод `Contains` позволяет сравнивать только с локально вычисляемым выражением; для сравнения с другим столбцом придется использовать метод `EF.Functions.Like`:

```
... where EF.Functions.Like (c.Description, "%" + c.Name + "%")
```

Метод `EF.Functions.Like` также позволяет выполнять более сложные сравнения (скажем, `LIKE 'abc%def%`).

Строковые сравнения < и > в EF Core

С помощью метода CompareTo типа string можно выполнять сравнение порядка для строк; он отображается на SQL-операции < и >:

```
dbContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

WHERE x IN (... , ... , ...) в EF Core

В EF Core операцию Contains можно применять к локальной коллекции внутри предиката фильтра. Например:

```
string[] chosenOnes = { "Tom", "Jay" };  
from c in dbContext.Customers  
where chosenOnes.Contains (c.Name)  
...
```

Это отображается на SQL-операцию IN, т.е.:

```
WHERE customer.Name IN ("Tom", "Jay")
```

Если локальная коллекция является массивом сущностей или элементов не-скалярных типов, то EF Core может взамен выпустить конструкцию EXISTS.

Take, TakeLast, Skip, SkipLast

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Количество элементов, которые необходимо выдать или пропустить	int

Операция Take выдает первые *n* элементов и отбрасывает остальные; операция Skip отбрасывает первые *n* элементов и выдает остальные. Эти два метода удобно применять вместе при реализации веб-страницы, позволяющей пользователю перемещаться по крупному набору записей. Например, пусть пользователь выполняет поиск в базе данных книг термина “mercury” (ртуть) и получает 100 совпадений. Приведенный ниже запрос возвращает первые 20 найденных книг:

```
IQueryable<Book> query = dbContext.Books  
.Where (b => b.Title.Contains ("mercury"))  
.OrderBy (b => b.Title)  
.Take (20);
```

Следующий запрос возвращает книги с 21-й по 40-ю:

```
IQueryable<Book> query = dbContext.Books  
.Where (b => b.Title.Contains ("mercury"))  
.OrderBy (b => b.Title)  
.Skip (20).Take (20);
```

Инфраструктура EF Core транслирует операции Take и Skip в функцию ROW_NUMBER для SQL Server 2005 и последующих версий или в подзапрос TOP для предшествующих версий SQL Server.

Методы `TakeLast` и `SkipLast` возвращают или пропускают последние n элементов.

Начиная с версии .NET 6, метод `Take` перегружен для приема переменной типа `Range`. Эта перегруженная версия может включать в себя функциональность всех четырех методов; например, вызов `Take(5..)` эквивалентен `Skip(5)`, а вызов `Take(..^5)` эквивалентен `SkipLast(5)`.

TakeWhile и SkipWhile

Аргумент	Тип
Исходная последовательность	<code>IEnumerable<TSource></code>
Предикат	<code>TSource => bool</code> или <code>(TSource, int) => bool</code>

Операция `TakeWhile` выполняет перечисление входной последовательности, выдавая элементы до тех пор, пока заданный предикат не станет равным `false`. Оставшиеся элементы игнорируются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100); // { 3, 5, 2 }
```

Операция `SkipWhile` выполняет перечисление входной последовательности, пропуская элементы до тех пор, пока заданный предикат не станет равным `false`. Оставшиеся элементы выдаются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100); // { 234, 4, 1 }
```

Операции `TakeWhile` и `SkipWhile` не транслируются в SQL и приводят к генерации исключения, когда присутствуют в запросе EF Core.

Distinct и DistinctBy

Операция `Distinct` возвращает входную последовательность, из которой удалены дубликаты. Дополнительно можно передавать специальный компаратор эквивалентности. Следующий запрос возвращает отличающиеся буквы в строке:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters); // H eloWr d
```

Методы LINQ можно вызывать прямо на строке, т.к. тип `string` реализует интерфейс `IEnumerable<char>`.

Метод `DistinctBy`, появившийся в версии .NET 6, позволяет указывать селектор ключей для применения перед выполнением сравнения эквивалентности. Результатом следующего выражения будет `{1, 2, 3}`:

```
new[] { 1.0, 1.1, 2.0, 2.1, 3.0, 3.1 }.DistinctBy (n => Math.Round (n, 0))
```

Выполнение проецирования

IEnumerable<TSource> → IEnumerable<TResult>

Метод	Описание	Эквиваленты в SQL
Select	Трансформирует каждый входной элемент с помощью заданного лямбда-выражения	SELECT
SelectMany	Трансформирует каждый входной элемент, а затем выравнивает и объединяет результатирующие подпоследовательности	INNER JOIN, LEFT OUTER JOIN, CROSS JOIN



При запрашивании базы данных операции Select и SelectMany являются наиболее универсальными конструкциями соединения; для локальных запросов операции Join и GroupJoin будут самыми эффективными конструкциями соединения.

Select

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Селектор результатов	TSource => TResult или (TSource, int) => TResult *

* Запрещено в EF Core.

Синтаксис запросов

select выражение-проекции

Реализация в Enumerable

```
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    foreach (TSource element in source)
        yield return selector (element);
}
```

Обзор

Посредством операции Select вы всегда получаете то же самое количество элементов, с которого начинали. Однако с помощью лямбда-функции каждый элемент может быть трансформирован произвольным образом.

Следующий запрос выбирает имена всех шрифтов, установленных на компьютере (через свойство `FontFamily.Families` из пространства имен `System.Drawing`):

```
IEnumerable<string> query = from f in FontFamily.Families
                                select f.Name;
foreach (string name in query) Console.WriteLine (name);
```

В этом примере конструкция `select` преобразует объект `FontFamily` в его имя. Ниже приведен эквивалент в виде лямбда-функции:

```
IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

Конструкции `Select` часто используются для проецирования в анонимные типы:

```
var query =
    from f in FontFamily.Families
    select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

Проектирование без трансформации иногда применяется в синтаксисе запросов с целью удовлетворения требования о том, что запрос должен заканчиваться конструкцией `select` или `group`. Следующий запрос извлекает шрифты, поддерживающие зачеркивание:

```
IEnumerable<FontFamily> query =
    from f in FontFamily.Families
    where f.IsStyleAvailable (FontStyle.Strikeout)
    select f;
foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

В таких случаях при переводе в текущий синтаксис компилятор опускает проецирование.

Индексированное проецирование

Выражение селектора может дополнительно принимать целочисленный аргумент, который действует в качестве индексатора, предоставляя выражение с позицией каждого элемента во входной последовательности. Прием работает только с локальными запросами:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names
    .Select ((s,i) => i + "=" + s);      // { "0=Tom", "1=Dick", ... }
```

Подзапросы `Select` и иерархии объектов

Для построения иерархии объектов подзапрос можно вкладывать внутрь конструкции `select`. В следующем примере возвращается коллекция, которая описывает каждый каталог в `Path.GetTempPath()`, с внутренней коллекцией файлов, хранящихся в каждом каталоге:

```
string tempPath = Path.GetTempPath();
DirectoryInfo[] dirs = new DirectoryInfo (tempPath).GetDirectories();
var query =
    from d in dirs
    where (d.Attributes & FileAttributes.System) == 0
    select new
    {
       DirectoryName = d.FullName,
        Created = d.CreationTime,
        Files = from f in d.GetFiles()
                where (f.Attributes & FileAttributes.Hidden) == 0
                select new { FileName = f.Name, f.Length, }
    };
};
```

```
foreach (var dirFiles in query)
{
    Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
    foreach (var file in dirFiles.Files)
        Console.WriteLine (" " + file.FileName + " Len: " + file.Length);
}
```

Внутреннюю часть запроса можно назвать *коррелированным подзапросом*. Подзапрос является коррелированным, если он ссылается на объект во внешнем запросе; в данном случае это d — каталог, содержимое которого перечисляется.



Подзапрос внутри Select позволяет отображать одну иерархию объектов на другую или отображать реляционную объектную модель на иерархическую объектную модель.

В локальных запросах подзапрос внутри Select приводит к дважды отложеному выполнению. В приведенном выше примере файлы не будут фильтроваться или проецироваться до тех пор, пока внутренний цикл foreach не начнет перечисление.

Подзапросы и соединения в EF Core

Проекции подзапросов эффективно функционируют в EF Core и могут использоваться для выполнения работы соединений в стиле SQL. Ниже показано, как извлечь для каждого заказчика имя и его дорогостоящие покупки:

```
var query =
    from c in dbContext.Customers
    select new {
        c.Name,
        Purchases = (from p in dbContext.Purchases
                     where p.CustomerID == c.ID && p.Price > 1000
                     select new { p.Description, p.Price })
                    .ToList()
    };
foreach (var namePurchases in query)
{
    Console.WriteLine ("Customer: " + namePurchases.Name);
    foreach (var purchaseDetail in namePurchases.Purchases)
        Console.WriteLine (" - $$$: " + purchaseDetail.Price);
}
```



Обратите внимание на использование `ToList` в подзапросе. Инфраструктура EF Core 3 не может создавать реализации `IQueryable` из результата подзапроса, когда этот подзапрос ссылается на `DbContext`. Проблема отслеживается командой разработчиков EF Core и в будущем выпуске может быть решена.



Такой стиль запроса идеально подходит для интерпретируемых запросов. Внешний запрос и подзапрос обрабатываются как единое целое, что позволяет предотвратить излишние обращения к серверу. Однако в случае локальных запросов это неэффективно, т.к. для получения нескольких соответствующих комбинаций потребуется выполнить перечисление каждой комбинации внешнего и внутреннего элементов. Более удачное решение для локальных запросов обеспечивают операции Join и GroupJoin, которые описаны в последующих разделах.

Приведенный выше запрос сопоставляет объекты из двух разных коллекций, и его можно считать “соединением”. Отличие между ним и обычным соединением базы данных (или подзапросом) заключается в том, что вывод не выравнивается в одиночный двумерный результирующий набор. Мы отображаем реляционные данные на иерархические, а не на плоские данные.

Вот как выглядит тот же самый запрос, упрощенный за счет применения навигационного свойства Purchases типа коллекции из сущностного класса Customer:

```
from c in dbContext.Customers
select new
{
    c.Name,
    Purchases = from p in c.Purchases           // Purchases имеет
                                                       // тип List<Purchase>
        where p.Price > 1000
        select new { p.Description, p.Price }
};
```

(При выполнении подзапроса с навигационным свойством в EF Core 3 вызов ToList не требуется.)

Оба запроса аналогичны левому внешнему соединению в SQL в том смысле, что при внешнем перечислении мы получаем всех заказчиков независимо от того, связаны с ними какие-либо покупки. Для эмуляции внутреннего соединения, посредством которого исключаются заказчики, не совершившие дорогостоящих покупок, необходимо добавить к коллекции покупок условие фильтрации:

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
    c.Name,
    Purchases = from p in c.Purchases
        where p.Price > 1000
        select new { p.Description, p.Price }
};
```

Запрос выглядит слегка неаккуратно из-за того, что один и тот же предикат (Price > 1000) записан дважды. Избежать подобного дублирования можно посредством конструкции let:

```

from c in dbContext.Customers
let highValueP = from p in c.Purchases
    where p.Price > 1000
    select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };

```

Представленный стиль запроса отличается определенной гибкостью. Например, изменив Any на Count, мы можем модифицировать запрос с целью извлечения только заказчиков, совершивших, по меньшей мере, две дорогостоящие покупки:

```

...
where highValueP.Count () >= 2
select new { c.Name, Purchases = highValueP };

```

Проектирование в конкретные типы

В приводимых до сих пор примерах мы создавали экземпляры анонимных типов в выходных данных. Кроме того, иногда удобно создавать экземпляры (обыкновенных) именованных классов, которые заполняются с помощью инициализаторов объектов. Такие классы могут включать специальную логику и передаваться между методами и сборками без использования информации о типах.

Типичным примером является специальная бизнес-сущность. Специальная бизнес-сущность — это просто разрабатываемый вами класс, который содержит ряд свойств, но спроектирован для скрытия низкоуровневых деталей (касающихся базы данных). Скажем, из классов бизнес-сущностей могут быть исключены поля внешних ключей. Предполагая, что специальные сущностные классы CustomerEntity и PurchaseEntity уже написаны, ниже показано, как можно было бы выполнить проектирование в них:

```

IQueryable<CustomerEntity> query =
    from c in dbContext.Customers
    select new CustomerEntity
    {
        Name = c.Name,
        Purchases =
            (from p in c.Purchases
            where p.Price > 1000
            select new PurchaseEntity {
                Description = p.Description,
                Value = p.Price
            })
        .ToList()
    };
// Обеспечить выполнение запроса, преобразовав вывод в более удобный список:
List<CustomerEntity> result = query.ToList();

```



Когда классы специальных бизнес-сущностей создаются для передачи данных между слоями в программе или между отдельными системами, они часто называются объектами передачи данных (data transfer object — DTO). Объекты передачи данных не содержат бизнес-логику.

Обратите внимание, что до сих пор мы не обязаны были применять операции Join или SelectMany. Причина в том, что мы поддерживали иерархическую форму данных, как показано на рис. 9.2. В LINQ часто удается избежать традиционного подхода SQL, при котором таблицы выравниваются в двухмерный результирующий набор.

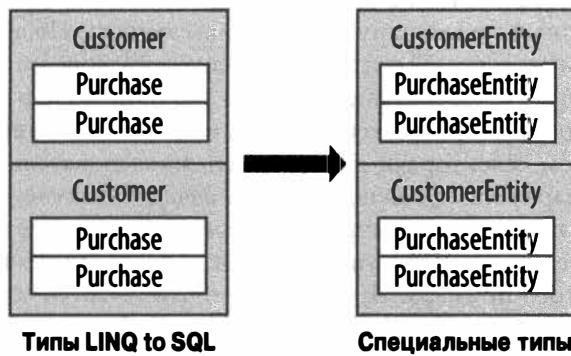


Рис. 9.2. Проектирование иерархии объектов

SelectMany

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Селектор результатов	TSource => IEnumerable<TResult> или (TSource, int) => IEnumerable<TResult> *

* Запрещено в EF Core.

Синтаксис запросов

```
from идентификатор1 in перечислимое_выражение1
from идентификатор2 in перечислимое_выражение2
...

```

Реализация в Enumerable

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>
    (IEnumerable<TSource> source,
     Func <TSource, IEnumerable<TResult>> selector)
{
    foreach (TSource element in source)
        foreach (TResult subElement in selector (element))
            yield return subElement;
}
```

Обзор

Операция SelectMany объединяет подпоследовательности в единую выходную последовательность.

Вспомните, что для каждого входного элемента операция Select выдает в точности один выходной элемент. Напротив, операция SelectMany выдает 0..n выходных элементов. Элементы 0..n берутся из подпоследовательности или дочерней последовательности, которую должно выпускать лямбда-выражение.

Операция SelectMany может использоваться для расширения дочерних последовательностей, выравнивания вложенных последовательностей и соединения двух коллекций в плоскую выходную последовательность. По аналогии с конвейерными лентами операция SelectMany помещает свежий материал на конвейерную ленту. Благодаря SelectMany каждый входной элемент является спусковым механизмом для подачи свежего материала. Свежий материал выпускается лямбда-выражением селектора и должен быть последовательностью. Другими словами, лямбда-выражение должно выдавать *дочернюю последовательность* для каждого входного элемента. Окончательным результатом будет объединение дочерних последовательностей, выпущенных для всех входных элементов.

Начнем с простого примера. Предположим, что имеется массив имён следующего вида:

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

который необходимо преобразовать в одиночную плоскую коллекцию слов:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", "Green"
```

Для решения задачи идеально подходит операция SelectMany, т.к. мы сопоставляем каждый входной элемент с переменным количеством выходных элементов. Все, что потребуется сделать — построить выражение селектора, которое преобразует каждый входной элемент в дочернюю последовательность. Для этого используется метод string.Split, который берет строку и разбивает ее на слова, выпуская результат в виде массива:

```
string testInputElement = "Anne Williams";
string[] childSequence = testInputElement.Split();
// childSequence содержит { "Anne", "Williams" };
```

Итак, ниже приведен запрос SelectMany и результат:

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());
foreach (string name in query)
    Console.Write (name + "|"); // Anne|Williams|John|Fred|Smith|Sue|Green|
```



Если заменить SelectMany операцией Select, то будет получен тот же самый результат, но в иерархической форме. Следующий запрос выдает последовательность строковых массивов, которая для своего перечисления требует вложенных операторов foreach:

```
IEnumerable<string[]> query =
    fullNames.Select (name => name.Split ());
foreach (string[] stringArray in query)
    foreach (string name in stringArray)
        Console.Write (name + "|");
```

Преимущество SelectMany в том, что выдается одиночная *плоская* результирующая последовательность.

Операция `SelectMany` поддерживается в синтаксисе запросов и вызывается с помощью *дополнительного генератора* — другими словами, дополнительной конструкции `from` в запросе. В синтаксисе запросов ключевое слово `from` исполняет две разных роли. В начале запроса оно вводит исходную переменную диапазона и входную последовательность. В *любом другом месте* запроса оно транслируется в операцию `SelectMany`. Ниже показан наш запрос, представленный в синтаксисе запросов:

```
IEnumerable<string> query =  
    from fullName in fullNames  
    from name in fullName.Split() // Транслируется в операцию SelectMany  
    select name;
```

Обратите внимание, что дополнительный генератор вводит новую переменную диапазона — в данном случае `name`. Тем не менее, старая переменная диапазона остается в области видимости, и мы можем работать с обеими переменными.

Множество переменных диапазона

В предыдущем примере переменные `name` и `fullName` остаются в области видимости до тех пор, пока не завершится запрос или не будет достигнута конструкция `into`. Расширенная область видимости упомянутых переменных представляет собой сценарий, в котором синтаксис запросов выигрывает у текущего синтаксиса.

Чтобы проиллюстрировать сказанное, мы можем взять предыдущий пример и включить `fullName` в финальную проекцию:

```
IEnumerable<string> query =  
    from fullName in fullNames  
    from name in fullName.Split()  
    select name + " came from " + fullName;  
  
// Anne came from Anne Williams  
// Williams came from Anne Williams  
// John came from John Fred Smith  
// ...
```

“За кулисами” компилятор должен предпринять ряд трюков, чтобы обеспечить доступ к обеим переменным. Хороший способ оценить ситуацию — попытаться написать тот же запрос в текущем синтаксисе. Это сложно! Задача еще более усложнится, если перед проецированием поместить конструкцию `where` или `orderby`:

```
from fullName in fullNames  
from name in fullName.Split()  
orderby fullName, name  
select name + " came from " + fullName;
```

Проблема в том, что операция `SelectMany` выдает плоскую последовательность дочерних элементов — в данном случае плоскую коллекцию слов. Исходный “внешний” элемент, из которого она поступает (`fullName`), утерян. Решение заключается в том, чтобы “передавать” внешний элемент с каждым дочерним элементом во временном анонимном типе:

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

Единственное изменение здесь заключается в том, что каждый дочерний элемент (name) помещается в оболочку анонимного типа, который также содержит его fullName. Это похоже на то, как распознается конструкция let. Ниже показано финальное преобразование в текущий синтаксис:

```
IEnumerable<string> query = fullNames
    .SelectMany (fName => fName.Split()
        .Select (name => new { name, fName } ))
    .OrderBy (x => x.fName)
    .ThenBy (x => x.name)
    .Select (x => x.name + " came from " + x.fName);
```

Мышление в терминах синтаксиса запросов

Как мы только что продемонстрировали, существуют веские причины применять синтаксис запросов, когда нужно работать с несколькими переменными диапазона. В таких случаях полезно не только использовать этот синтаксис, но также и думать непосредственно в его терминах.

При написании дополнительных генераторов применяются два базовых шаблона. Первый из них — *расширение и выравнивание подпоследовательностей*. Для этого в дополнительном генераторе производится обращение к свойству или методу с существующей переменной диапазона. Мы поступали так в предыдущем примере:

```
from fullName in fullNames
from name in fullName.Split()
```

Здесь мы расширили перечисление фамилий до перечисления слов. Аналогичный запрос EF Core производится, когда необходимо развернуть навигационные свойства типа коллекций. Следующий запрос выводит список всех заказчиков вместе с их покупками:

```
IEnumerable<string> query = from c in dbContext.Customers
                                from p in c.Purchases
                                select c.Name + " bought a " + p.Description;

// Tom bought a Bike
// Tom bought a Holiday
// Dick bought a Phone
// Harry bought a Car
// ...
```

Здесь мы расширили каждого заказчика в подпоследовательность покупок.

Второй шаблон предусматривает выполнение *декартова произведения*, или *перекрестного соединения*, при котором каждый элемент одной последовательности сопоставляется с каждым элементом другой последовательности. Чтобы сделать это, потребуется ввести генератор, выражение селектора которого возвращает последовательность, не связанную с какой-то переменной диапазона:

```
int[] numbers = { 1, 2, 3 }; string[] letters = { "a", "b" };
IQueryable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;
// Результат: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

Такой стиль запроса является основой *соединений* в стиле **SelectMany**.

Выполнение соединений с помощью **SelectMany**

Операцию **SelectMany** можно использовать для соединения двух последовательностей, просто отфильтровывая результаты векторного произведения. Например, предположим, что необходимо сопоставить игроков друг с другом в игре. Мы можем начать следующим образом:

```
string[] players = { "Tom", "Jay", "Mary" };
IQueryable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;
// Результат: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
//               "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
//               "Mary vs Tom", "Mary vs Jay", "Mary vs Mary" }
```

Запрос читается так: “Для любого игрока выполнить итерацию по каждому игроку, выбирая игрока 1 против игрока 2”. Хотя мы получаем то, что запросили (перекрестное соединение), результаты бесполезны до тех пор, пока не будет добавлен фильтр:

```
IQueryable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo(name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;
// Результат: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

Предикат фильтра образует *условие соединения*. Наш запрос можно назвать *неэквисоединением*, потому что в условии соединения операция эквивалентности не применяется.

SelectMany в EF Core

Операция **SelectMany** в EF Core может выполнять перекрестные соединения, неэквисоединения, внутренние соединения и левые внешние соединения. Как и **Select**, ее можно применять с предопределенными ассоциациями и произвольными отношениями. Отличие в том, что операция **SelectMany** возвращает плоский, а не иерархический результирующий набор.

Перекрестное соединение EF Core записывается точно так же, как в предыдущем разделе. Следующий запрос сопоставляет каждого заказчика с каждой покупкой (перекрестное соединение):

```
var query = from c in dbContext.Customers
            from p in dbContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

Однако более типичной ситуацией является сопоставление заказчиков только с их собственными покупками. Это достигается добавлением конструкции `where` с предикатом соединения. В результате получается стандартное эквисоединение в стиле SQL:

```
var query = from c in dataContext.Customers
             from p in dataContext.Purchases
             where c.ID == p.CustomerID
             select c.Name + " bought a " + p.Description;
```



Такой запрос хорошо транслируется в SQL. В следующем разделе будет показано, как его расширить для поддержки внешних соединений. Переписывание запросов подобного рода с использованием LINQ-операции `Join` в действительности делает их *менее* расширяемыми — в таком смысле язык LINQ противоположен SQL.

При наличии в сущностях навигационных свойств типа коллекций тот же самый запрос можно выразить путем развертывания подколлекции вместо фильтрации результатов векторного произведения:

```
from c in dbContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

Преимущество заключается в том, что мы устранием предикат соединения. Мы перешли от фильтрации векторного произведения к развертыванию и выравниванию.

Для дополнительной фильтрации к такому запросу можно добавлять конструкции `where`. Например, если нужны только заказчики, имена которых начинаются на “T”, фильтрацию можно производить так:

```
from c in dbContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

Приведенный запрос EF Core будет работать в равной степени хорошо, если переместить конструкцию `where` на одну строку ниже. Однако если это локальный запрос, то перемещение конструкции `where` вниз может сделать его менее эффективным. Для локальных запросов фильтрация должна выполняться *перед* соединением.

С помощью дополнительных конструкций `from` в полученную смесь можно вводить новые таблицы. Например, если каждая запись о покупке имеет дочерние строки деталей, то можно было бы построить плоский результирующий набор заказчиков с их покупками и деталями по каждой покупке:

```
from c in dbContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.Detail };
```

Каждая конструкция `from` вводит новую дочернюю таблицу. Чтобы включить данные из родительской таблицы (через навигационное свойство), не нужно добавлять конструкцию `from` — необходимо лишь перейти на это свойство. Скажем, если с каждым заказчиком связан продавец, имя которого требуется запросить, можно поступить следующим образом:

```
from c in dbContext.Customers  
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

В данном случае операция `SelectMany` не используется, т.к. отсутствуют подколлекции, подлежащие выравниванию. Родительское навигационное свойство возвращает одиночный элемент.

Выполнение внешних соединений с помощью `SelectMany`

Как было показано ранее, подзапрос `Select` выдает результат, аналогичный левому внешнему соединению:

```
from c in dbContext.Customers  
select new {  
    c.Name,  
    Purchases = from p in c.Purchases  
                where p.Price > 1000  
                select new { p.Description, p.Price }  
};
```

В приведенном примере каждый внешний элемент (заказчик) включается независимо от того, совершились ли им какие-то покупки. Но предположим, что запрос переписан с применением операции `SelectMany`, а потому вместо иерархического результирующего набора можно получить одиночную плоскую коллекцию:

```
from c in dbContext.Customers  
from p in c.Purchases  
where p.Price > 1000  
select new { c.Name, p.Description, p.Price };
```

В процессе выравнивания запроса мы перешли на внутреннее соединение: теперь включаются только такие заказчики, которые имеют одну или более дорогостоящих покупок. Чтобы получить левое внешнее соединение с плоским результирующим набором, мы должны применить к внутренней последовательности операцию запроса `DefaultIfEmpty`. Этот метод возвращает последовательность с единственным элементом `null`, когда входная последовательность не содержит элементов. Ниже показан такой запрос с опущенным предикатом цены:

```
from c in dbContext.Customers  
from p in c.Purchases.DefaultIfEmpty()  
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

Данный запрос успешно работает с EF Core, возвращая всех заказчиков, даже если они вообще не совершали покупок. Но в случае его запуска как локального запроса произойдет аварийный отказ, потому что когда `p` равно `null`, обращения `p.Description` и `p.Price` генерируют исключение `NullReferenceException`. Мы можем сделать наш запрос надежным в обоих сценариях следующим образом:

```

from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
    c.Name,
    Descript = p == null ? null : p.Description,
    Price = p == null ? (decimal?) null : p.Price
};

```

Теперь давайте займемся фильтром цены. Использовать конструкцию `where`, как мы поступали ранее, не получится, поскольку она выполняется *после* `DefaultIfEmpty`:

```

from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...

```

Корректное решение предусматривает сращивание конструкции `Where` *перед* `DefaultIfEmpty` с подзапросом:

```

from c in dbContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
    c.Name,
    Descript = p == null ? null : p.Description,
    Price = p == null ? (decimal?) null : p.Price
};

```

Инфраструктура EF Core транслирует такой запрос в левое внешнее соединение. Это эффективный шаблон для написания запросов подобного рода.



Если вы привыкли к написанию внешних соединений на языке SQL, то можете не заметить более простой вариант с подзапросом `Select` для такого стиля запросов, а отдать предпочтение неудобному, но знакомому подходу, ориентированному на SQL, с плоским результатирующим набором. Иерархический результатирующий набор из подзапроса `Select` часто лучше подходит для запросов с внешними соединениями, потому что в таком случае отсутствуют дополнительные значения `null`, с которыми пришлось бы иметь дело.

Выполнение соединения

Метод	Описание	Эквиваленты в SQL
Join	Применяет стратегию поиска для сопоставления элементов из двух коллекций, выпуская плоский результатирующий набор	INNER JOIN
GroupJoin	Похож на <code>Join</code> , но выпускает <i>иерархический</i> результатирующий набор	INNER JOIN, LEFT OUTER JOIN
Zip	Перечисляет две последовательности за раз (подобно застежке-молнии (<i>zipper</i>)), применяя функцию к каждой паре элементов	Генерируется исключение

Join и GroupJoin

IEnumerable<TOuter>, IEnumerable<TInner> → IEnumerable<TResult>

Аргументы Join

Аргумент	Тип
Внешняя последовательность	IEnumerable<TOuter>
Внутренняя последовательность	IEnumerable<TInner>
Внешний селектор ключей	TOuter => TKey
Внутренний селектор ключей	TInner => TKey
Селектор результатов	(TOuter, TInner) => TResult

Аргументы GroupJoin

Аргумент	Тип
Внешняя последовательность	IEnumerable<TOuter>
Внутренняя последовательность	IEnumerable<TInner>
Внешний селектор ключей	TOuter => TKey
Внутренний селектор ключей	TInner => TKey
Селектор результатов	(TOuter, IEnumerable<TInner>) => TResult

Синтаксис запросов

```
from внешняя-переменная in внешнее-перечисление
join внутренняя-переменная in внутреннее-перечисление
    on внешнее-выражение-ключей equals внутреннее-выражение-ключей
[ into идентификатор ]
```

Обзор

Операции Join и GroupJoin объединяют две входные последовательности в единственную выходную последовательность. Операция Join выпускает плоский вывод, а GroupJoin — иерархический.

Операции Join и GroupJoin поддерживают стратегию, альтернативную операциям Select и SelectMany. Преимущество Join и GroupJoin связано с эффективным выполнением на локальных коллекциях в памяти, т.к. они сначала загружают внутреннюю последовательность в объект Lookup с ключами, устранив необходимость в повторяющемся перечислении по всем внутренним элементам. Их недостаток в том, что они предлагают эквивалент только для внутренних и левых внешних соединений; перекрестные соединения и неэквисоединения по-прежнему должны делаться с помощью Select/SelectMany. В запросах EF Core операции Join и GroupJoin не имеют реальных преимуществ перед Select и SelectMany.

В табл. 9.1 приведена сводка по различиям между стратегиями соединения.

Таблица 9.1. Стратегии соединения

Стратегия	Форма результатов	Эффективность локальных запросов	Внутренние соединения	Левые внешние соединения	Перекрестные соединения	Незкви-соединения
Select + SelectMany	Плоская	Низкая	Да	Да	Да	Да
Select + Select	Вложенная	Низкая	Да	Да	Да	Да
Join	Плоская	Хорошая	Да	-	-	-
GroupJoin	Вложенная	Хорошая	Да	Да	-	-
GroupJoin + SelectMany	Плоская	Хорошая	Да	Да	-	-

Join

Операция Join выполняет внутреннее соединение, выпуская плоскую выходную последовательность.

Следующий запрос выводит список всех заказчиков вместе с их покупками, не используя навигационное свойство:

```
IQueryable<string> query =  
    from c in dbContext.Customers  
    join p in dbContext.Purchases on c.ID equals p.CustomerID  
    select c.Name + " bought a " + p.Description;
```

Результаты совпадают с теми, которые были бы получены из запроса в стиле SelectMany:

```
Tom bought a Bike  
Tom bought a Holiday  
Dick bought a Phone  
Harry bought a Car
```

Чтобы увидеть преимущество операции Join перед SelectMany, мы должны преобразовать запрос в локальный. В целях демонстрации мы сначала скопируем всех заказчиков и покупки в массивы, после чего выполним запросы к этим массивам:

```
Customer[] customers = dbContext.Customers.ToArray();  
Purchase[] purchases = dbContext.Purchases.ToArray();  
var slowQuery = from c in customers  
                 from p in purchases where c.ID == p.CustomerID  
                 select c.Name + " bought a " + p.Description;  
  
var fastQuery = from c in customers  
                 join p in purchases on c.ID equals p.CustomerID  
                 select c.Name + " bought a " + p.Description;
```

Хотя оба запроса выдают одинаковые результаты, запрос `Join` заметно быстрее, потому что его реализация в классе `Enumerable` предварительно загружает внутреннюю коллекцию (`purchases`) в объект `Lookup` с ключами.

Синтаксис запросов для конструкции `join` может быть записан в общем случае так:

```
join внутренняя-переменная in внутренняя-последовательность
    on внешнее-выражение-ключей equals внутреннее-выражение-ключей
```

Операции соединений в LINQ проводят различие между *внутренней последовательностью* и *внешней последовательностью*. Вот что они означают с точки зрения синтаксиса.

- *Внешняя последовательность* — это входная последовательность (в данном случае `customers`).
- *Внутренняя последовательность* — это введенная вами новая коллекция (в данном случае `purchases`).

Операция `Join` выполняет внутренние соединения, а значит заказчики, не имеющие связанных с ними покупок, из вывода исключаются. При внутренних соединениях внутреннюю и внешнюю последовательности в запросе можно менять местами и по-прежнему получать те же самые результаты:

```
from p in purchases                                // p теперь внешняя последовательность
join c in customers on p.CustomerID equals c.ID    // c теперь внутренняя
                                                               // последовательность
...

```

В запрос можно добавлять дополнительные конструкции `join`. Если, например, каждая запись о покупке имеет один или более элементов, тогда выполнить соединение элементов покупок можно следующим образом:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID      // первое соединение
join pi in purchaseItems on p.ID equals pi.PurchaseID // второе
                                                               // соединение
...

```

Здесь `purchases` действует в качестве *внутренней* последовательности в первом соединении и в качестве *внешней* — во втором. Получить те же самые результаты (неэффективным способом) можно с применением вложенных операторов `foreach`:

```
foreach (Customer c in customers)
    foreach (Purchase p in purchases)
        if (c.ID == p.CustomerID)
            foreach (PurchaseItem pi in purchaseItems)
                if (p.ID == pi.PurchaseID)
                    Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

В синтаксисе запросов переменные из более ранних соединений остаются в области видимости — точно так происходит и в запросах стиля `SelectMany`. Кроме того, между конструкциями `join` разрешено вставлять конструкции `where` и `let`.

Выполнение соединений по нескольким ключам

Можно выполнять соединение по нескольким ключам с помощью анонимных типов:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                           equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
...
```

Чтобы прием работал, два анонимных типа должны быть идентично структурированными. Компилятор затем реализует каждый из них с помощью одного и того же внутреннего типа, делая соединяемые ключи совместимыми.

Выполнение соединений в текущем синтаксисе

Показанное ниже соединение в синтаксисе запросов:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

можно выразить с помощью текущего синтаксиса:

```
customers.Join (
    purchases,                      // внешняя коллекция
    c => c.ID,                      // внутренняя коллекция
    p => p.CustomerID,             // внешний селектор ключей
    (c, p) => new                  // внутренний селектор ключей
        { c.Name, p.Description, p.Price } // селектор результатов
);
```

Выражение селектора результатов в конце создает каждый элемент в выходной последовательности. Если перед проецированием присутствуют дополнительные конструкции, такие как `orderby` в данном примере:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

тогда для представления запроса в текущем синтаксисе потребуется создать временный анонимный тип внутри селектора результатов. Такой прием сохраняет `c` и `p` в области видимости, следуя соединению:

```
customers.Join (
    purchases,                      // внешняя коллекция
    c => c.ID,                      // внутренняя коллекция
    p => p.CustomerID,             // внешний селектор ключей
    (c, p) => new { c, p } )       // селектор результатов
    .OrderBy (x => x.p.Price)
    .Select (x => x.c.Name + " bought a " + x.p.Description);
```

При выполнении соединений синтаксис запросов обычно предпочтительнее; он требует менее кропотливой работы.

GroupJoin

Операция GroupJoin делает то же самое, что и Join, но вместо плоского результата выдает иерархический результат, сгруппированный по каждому внешнему элементу. Она также позволяет выполнять левые внешние соединения. В настоящее время операция GroupJoin в EF Core не поддерживается.

Синтаксис запросов для GroupJoin такой же, как и для Join, но за ним следует ключевое слово `into`.

Вот простейший пример, в котором используется локальный запрос:

```
Customer[] customers = dbContext.Customers.ToArray();
Purchase[] purchases = dbContext.Purchases.ToArray();

IEnumerable<IEnumerable<Purchase>> query =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select custPurchases; // custPurchases является последовательностью
```



Конструкция `into` транслируется в операцию GroupJoin, только когда она появляется непосредственно после конструкции `join`. После конструкции `select` или `group` она означает *продолжение запроса*. Указанные два сценария использования ключевого слова `into` значительно отличаются, хотя и обладают одной общей характеристикой: в обоих случаях вводится новая переменная диапазона.

Результатом будет последовательность последовательностей, для которой можно выполнить перечисление:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
    foreach (Purchase p in purchaseSequence)
        Console.WriteLine (p.Description);
```

Тем не менее, это не особенно полезно, т.к. `purchaseSequence` не имеет ссылок на заказчика. Чаще всего следует поступать так:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Результаты будут такими же, как у приведенного ниже (незэффективного) подзапроса `Select`:

```
from c in customers
select new
{
    CustName = c.Name,
    custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

По умолчанию операция GroupJoin является эквивалентом левого внешнего соединения. Чтобы получить внутреннее соединение, посредством которого исключаются заказчики без покупок, понадобится реализовать фильтрацию по `custPurchases`:

```
from c in customers join p in purchases on c.ID equals p.CustomerID  
into custPurchases  
where custPurchases.Any()  
select ...
```

Конструкции после `into` в `GroupJoin` оперируют на *подпоследовательностях* внутренних дочерних элементов, а не на *индивидуальных* дочерних элементах. Это значит, что для фильтрации отдельных покупок необходимо вызвать операцию `Where` перед соединением:

```
from c in customers  
join p in purchases.Where (p2 => p2.Price > 1000)  
on c.ID equals p.CustomerID  
into custPurchases ...
```

С помощью операции `GroupJoin` можно конструировать лямбда-выражения, как это делается посредством `Join`.

Плоские внешние соединения

Когда требуется и внешнее соединение, и плоский результирующий набор, возникает дилемма. Операция `GroupJoin` обеспечивает внешнее соединение, а `Join` дает плоский результирующий набор. Решение заключается в том, чтобы сначала вызвать `GroupJoin`, затем метод `DefaultIfEmpty` на каждой дочерней последовательности и, наконец, метод `SelectMany` на результате:

```
from c in customers  
join p in purchases on c.ID equals p.CustomerID into custPurchases  
from cp in custPurchases.DefaultIfEmpty()  
select new  
{  
    CustName = c.Name,  
    Price = cp == null ? (decimal?) null : cp.Price  
};
```

Метод `DefaultIfEmpty` возвращает последовательность с единственным значением `null`, если подпоследовательность покупок пуста. Вторая конструкция `from` транслируется в вызов метода `SelectMany`. В такой роли она *развертывает и выравнивает* все подпоследовательности покупок, объединяя их в единую последовательность элементов покупок.

Выполнение соединений с помощью объектов `Lookup`

Методы `Join` и `GroupJoin` в `Enumerable` работают в два этапа. Во-первых, они загружают внутреннюю последовательность в объект `Lookup`. Во-вторых, они запрашивают внешнюю последовательность в комбинации с объектом `Lookup`.

Объект `Lookup` — это последовательность групп, в которую можно получать доступ напрямую по ключу. По-другому его следует воспринимать как словарь последовательностей — словарь, который способен принимать множество элементов для каждого ключа (иногда его называют *мультисловарем*). Объекты `Lookup` предназначены только для чтения и определены в соответствии со следующим интерфейсом:

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



Операции соединений (как и все остальные операции, выдающие последовательности) поддерживают семантику отложенного или ленивого выполнения. Это значит, что объект Lookup не будет создан до тех пор, пока вы не начнете перечисление выходной последовательности (и тогда сразу же строится целый объект Lookup).

Имея дело с локальными коллекциями, в качестве альтернативы применению операций соединения объекты Lookup можно создавать и запрашивать вручную. Такой подход обладает парой преимуществ:

- один и тот же объект Lookup можно многократно использовать во множестве запросов — равно как и в обычном императивном коде;
- выдача запросов к объекту Lookup — отличный способ понять, каким образом работают операции Join и GroupJoin.

Объект Lookup создается с помощью расширяющего метода ToLookup. Следующий код загружает все покупки в объект Lookup — с ключами в виде их идентификаторов CustomerID:

```
ILookup<int?, Purchase> purchLookup =  
    purchases.ToLookup (p => p.CustomerID, p => p);
```

Первый аргумент выбирает ключ, а второй — объекты, которые должны загружаться в качестве значений в Lookup.

Чтение объекта Lookup довольно похоже на чтение словаря за исключением того, что индексатор возвращает *последовательность* соответствующих элементов, а не *одиночный* элемент. Приведенный ниже код перечисляет все покупки, совершенные заказчиком с идентификатором 1:

```
foreach (Purchase p in purchLookup [1])  
    Console.WriteLine (p.Description);
```

При наличии объекта Lookup можно написать запросы SelectMany/Select, которые выполняются настолько же эффективно, как запросы Join/GroupJoin. Операция Join эквивалентна применению метода SelectMany на объекте Lookup:

```
from c in customers  
from p in purchLookup [c.ID]  
select new { c.Name, p.Description, p.Price };  
  
// Tom Bike 500  
// Tom Holiday 2000  
// Dick Bike 600  
// Dick Phone 300  
// ...
```

Добавление вызова метода DefaultIfEmpty делает этот запрос внутренним соединением:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
select new {
    c.Name,
    Description = p == null ? null : p.Description,
    Price = p == null ? (decimal?) null : p.Price
};
```

Использование GroupJoin эквивалентно чтению объекта Lookup внутри проекции:

```
from c in customers
select new {
    CustName = c.Name,
    CustPurchases = purchLookup [c.ID]
};
```

Реализация в Enumerable

Ниже представлена простейшая допустимая реализация Enumerable.Join, не включающая проверку на предмет равенства null:

```
public static IEnumerable <TResult> Join
    <TOuter, TInner, TKey, TResult> (
        this IEnumerable <TOuter>      outer,
        IEnumerable <TInner>          inner,
        Func <TOuter, TKey>          outerKeySelector,
        Func <TInner, TKey>          innerKeySelector,
        Func <TOuter, TInner, TResult> resultSelector)
{
    ILookup < TKey, TInner > lookup = inner.ToLookup (innerKeySelector);
    return
        from outerItem in outer
        from innerItem in lookup [outerKeySelector (outerItem)]
        select resultSelector (outerItem, innerItem);
}
```

Реализация GroupJoin похожа на реализацию Join, но только проще:

```
public static IEnumerable <TResult> GroupJoin
    <TOuter, TInner, TKey, TResult> (
        this IEnumerable <TOuter>      outer,
        IEnumerable <TInner>          inner,
        Func <TOuter, TKey>          outerKeySelector,
        Func <TInner, TKey>          innerKeySelector,
        Func <TOuter, IEnumerable<TInner>, TResult> resultSelector)
{
    ILookup < TKey, TInner > lookup = inner.ToLookup (innerKeySelector);
    return
        from outerItem in outer
        select resultSelector
            (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

Операция Zip

`IEnumerable<TFirst>, IEnumerable<TSecond> → IEnumerable<TResult>`

Операция `Zip` выполняет перечисление двух последовательностей за раз (подобно застежке-молнии (`zipper`)) и возвращает последовательность, основанную на применении некоторой функции к каждой паре элементов. Например, следующий код:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

выдает последовательность с такими элементами:

```
3=three
5=five
7=seven
```

Избыточные элементы в любой из входных последовательностей игнорируются. Операция `Zip` не поддерживается инфраструктурой EF Core.

Упорядочение

`IEnumerable<TSource> → IOrderedEnumerable<TSource>`

Метод	Описание	Эквиваленты в SQL
<code>OrderBy, ThenBy</code>	Сортируют последовательность в возрастающем порядке	<code>ORDER BY...</code>
<code>OrderByDescending, ThenByDescending</code>	Сортируют последовательность в убывающем порядке	<code>ORDER BY... DESC</code>
<code>Reverse</code>	Возвращает последовательность в обратном порядке	Генерируется исключение

Операции упорядочения возвращают те же самые элементы, но в другом порядке.

`OrderBy, OrderByDescending, ThenBy, ThenByDescending`

Аргументы `OrderBy` и `OrderByDescending`

Аргумент	Тип
Входная последовательность	<code>IEnumerable<TSource></code>
Селектор ключей	<code>TSource => TKey</code>

Возвращаемый тип: `IOrderedEnumerable<TSource>`

Аргументы ThenBy и ThenByDescending

Аргумент	Тип
Входная последовательность	IOrderedEnumerable<TSource>
Селектор ключей	TSource => TKey

Синтаксис запросов

```
orderby выражение1 [descending] [, выражение2 [descending] ... ]
```

Обзор

Операция OrderBy возвращает отсортированную версию входной последовательности, используя выражение keySelector для выполнения сравнений. Следующий запрос выдает последовательность имен в алфавитном порядке:

```
IEnumerable<string> query = names.OrderBy (s => s);
```

А здесь имена сортируются по их длине:

```
IEnumerable<string> query = names.OrderBy (s => s.Length);
```

```
// Результат: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

Относительный порядок элементов с одинаковыми ключами сортировки (в данном случае Jay/Tom и Mary/Dick) не определен, если только не добавить операцию ThenBy:

```
IEnumerable<string> query = names.OrderBy (s => s.Length).ThenBy (s => s);
```

```
// Результат: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

Операция ThenBy переупорядочивает лишь элементы, которые имеют один и тот же ключ сортировки из предшествующей сортировки. Допускается выстраивать в цепочку любое количество операций ThenBy. Следующий запрос сортирует сначала по длине, затем по второму символу и, наконец, по первому символу:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

Ниже показан эквивалент в синтаксисе запросов:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```



Приведенная далее вариация некорректна — в действительности она будет упорядочивать сначала по s[1] и затем по s.Length (в случае запроса к базе данных она будет упорядочивать только по s[1] и отбрасывать первое упорядочение):

```
from s in names
orderby s.Length
orderby s[1]
...

```

Язык LINQ также предлагает операции `OrderByDescending` и `ThenByDescending`, которые делают то же самое, выдавая результаты в обратном порядке. Следующий запрос EF Core извлекает покупки в убывающем порядке цен, выстраивая покупки с одинаковой ценой в алфавитном порядке:

```
dbContext.Purchases.OrderByDescending (p => p.Price)
    .ThenBy (p => p.Description);
```

А вот он в синтаксисе запросов:

```
from p in dbContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Компараторы и сопоставления

В локальном запросе объекты селекторов ключей самостоятельно определяют алгоритм упорядочения через свою стандартную реализацию интерфейса `IComparable` (см. главу 7). Переопределить алгоритм сортировки можно путем передачи объекта реализации `IComparer`. Показанный ниже запрос выполняет сортировку, нечувствительную к регистру:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Передача компаратора не поддерживается ни в синтаксисе запросов, ни в EF Core. При запросе базы данных алгоритм сравнения определяется сопоставлением участящего столбца. Если сопоставление чувствительно к регистру, то нечувствительную к регистру сортировку можно затребовать вызовом метода `ToUpper` в селекторе ключей:

```
from p in dbContext.Purchases
orderby p.Description.ToUpper()
select p;
```

IOrderedEnumerable и IOrderedQueryable

Операции упорядочения возвращают специальные подтипы `IEnumerable<T>`. Операции упорядочения в классе `Enumerable` возвращают тип `IOrderedEnumerable<TSource>`, а операции упорядочения в классе `Queryable` — тип `IOrderedQueryable<TSource>`. Упомянутые подтипы позволяют с помощью последующей операции `ThenBy` уточнять, а не заменять существующее упорядочение.

Дополнительные члены, которые эти подтипы определяют, не открыты публично, так что они представляются как обычные последовательности. Тот факт, что они являются разными типами, вступает в игру при постепенном построении запросов:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

Если взамен объявить переменную `query1` как имеющую тип `IEnumerable<string>`, тогда вторая строка не скомпилируется — операция `ThenBy` требует на входе тип `IOrderedEnumerable<string>`. Чтобы не переживать по такому поводу, переменные диапазона можно типизировать неявно:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Однако неявная типизация и сама может создать проблемы. Следующий код не скомпилируется:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3); // Ошибка на этапе компиляции
```

В качестве типа переменной `query` компилятор выводит `IOrderedEnumerable<string>`, основываясь на типе выходной последовательности операции `OrderBy`. Тем не менее, операция `Where` в следующей строке возвращает обычную реализацию `IEnumerable<string>`, которая не может быть присвоена `query`. Обойти проблему можно либо за счет явной типизации, либо путем вызова метода `AsEnumerable` после `OrderBy`:

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3); // Компилируется
```

Эквивалентом `AsEnumerable` в интерпретируемых запросах является метод `AsQueryable`.

Группирование

`IEnumerable<TSource> → IEnumerable<IGrouping< TKey, TElement >>`

Метод	Описание	Эквиваленты в SQL
<code>GroupBy</code>	Группирует последовательность в подпоследовательности	GROUP BY
<code>Chunk</code>	Группирует последовательность в массивы фиксированного размера	

GroupBy

Аргумент	Тип
Входная последовательность	<code>IEnumerable<TSource></code>
Селектор ключей	<code>TSource => TKey</code>
Селектор элементов (необязательный)	<code>TSource => TElement</code>
Компаратор (необязательный)	<code>IEqualityComparer< TKey ></code>

Синтаксис запросов

`group выражение-элементов by выражение-ключей`

Обзор

Операция `GroupBy` организует плоскую входную последовательность в последовательность *групп*. Например, приведенный ниже код организует все файлы в каталоге `Path.GetTempPath()` по их расширениям:

```
string[] files = Directory.GetFiles ("c:\\temp");
IEnumerable<IGrouping<string, string>> query =
    files.GroupBy (file => Path.GetExtension (file));
```

Если неявная типизация удобнее, то можно записать так:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

А вот как выполнить перечисление результата:

```
foreach (IGrouping<string, string> grouping in query)
{
    Console.WriteLine ("Extension: " + grouping.Key);
    foreach (string filename in grouping)
        Console.WriteLine ("    -- " + filename);
}

// Extension: .pdf
//   -- chapter03.pdf
//   -- chapter04.pdf
// Extension: .doc
//   -- todo.doc
//   -- menu.doc
//   -- Copy of menu.doc
// ...
```

Метод `Enumerable.GroupBy` работает путем чтения входных элементов во временный словарь списков, так что все элементы с одинаковыми ключами попадают в один и тот же подсписок. Затем выдается последовательность групп. Группа — это последовательность со свойством `Key`:

```
public interface IGrouping < TKey, TElement > : IEnumerable< TElement >,
    IEnumerable
{
    TKey Key { get; } // Ключ применяется к подпоследовательности
                     // как к единому целому
}
```

По умолчанию элементы в каждой группе являются нетрансформированными входными элементами, если только не указан аргумент `elementSelector`. Следующий код проецирует каждый входной элемент в верхний регистр:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

Аргумент `elementSelector` не зависит от `keySelector`. В данном случае это означает, что ключ каждой группы по-прежнему представлен в первоначальном регистре:

```
Extension: .pdf
-- CHAPTER03.PDF
-- CHAPTER04.PDF
Extension: .doc
-- TODO.DOC
```

Обратите внимание, что подколлекции не выдаются в алфавитном порядке ключей. Операция `GroupBy` просто группирует; она не выполняет *сортировку*. В действительности она предохраняет исходное упорядочение.

Для сортировки потребуется добавить операцию OrderBy:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
    .OrderBy (grouping => grouping.Key);
```

Операция GroupBy имеет простую и прямую трансляцию в синтаксис запросов:

```
group выражение-элементов by выражение-ключей
```

Ниже приведен наш пример, представленный в синтаксисе запросов:

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

Как и select, конструкция group “заканчивает” запрос — если только не была добавлена конструкция продолжения запроса:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

Продолжения запросов часто удобны в запросах group by. Следующий запрос отфильтровывает группы, которые содержат менее пяти файлов:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count () >= 5
select grouping;
```



Конструкция where после group by является эквивалентом конструкции HAVING в SQL. Она применяется к каждой подпоследовательности или группе как к единому целому, а не к ее индивидуальным элементам.

Иногда интересует только результат агрегирования на группах, а потому подпоследовательности можно отбросить:

```
string [] votes = { "Dogs", "Cats", "Cats", "Dogs", "Dogs" };
IEnumerable<string> query = from vote in votes
                                group vote by vote into g
                                orderby g.Count () descending
                                select g.Key;
string winner = query.First(); // Dogs
```

GroupBy в EF Core

При запрашивании базы данных группирование работает аналогичным образом. Однако если есть настроенные навигационные свойства, то вы обнаружите, что потребность в группировании возникает менее часто, чем при работе со стандартным языком SQL. Например, для выборки заказчиков с не менее чем двумя покупками группирование не понадобится; запрос может быть записан так:

```
from c in dbContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

Примером, когда может использоваться группирование, служит вывод списка итоговых продаж по годам:

```
from p in dbContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
    Year      = salesByYear.Key,
    TotalValue = salesByYear.Sum()
};
```

По сравнению с конструкцией GROUP BY в SQL группирование в LINQ отличается большей мощностью — вы можете извлечь все строки с деталями покупок безо всякого агрегирования:

```
from p in dbContext.Purchases
group p by p.Date.Year
```

Однако в EF Core такой прием не работает. Проблему легко обойти за счет вызова метода AsEnumerable прямо перед группированием, чтобы группирование происходило на стороне клиента. Это будет не менее эффективным до тех пор, пока любая фильтрация выполняется перед группированием, так что из сервера извлекаются только те данные, которые необходимы.

Еще одно отклонение от традиционного языка SQL связано с отсутствием обязательного проектирования переменных или выражений, участвующих в группировании или сортировке.

Группирование по нескольким ключам

Можно группировать по составному ключу с применением анонимного типа:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

Специальные компараторы эквивалентности

В локальном запросе для изменения алгоритма сравнения ключей методу GroupBy можно передавать специальный компаратор эквивалентности. Однако такое действие требуется редко, потому что модификации выражения селектора ключей обычно вполне достаточно. Например, следующий запрос создает группирование, нечувствительное к регистру:

```
group n by n.ToUpper()
```

Chunk

IEnumerable<TSource> → IEnumerable<TElement []>

Аргумент	Тип
Входная последовательность	IEnumerable<TSource>
Размер	int

Появившаяся в версии .NET 6 операция `Chunk` группирует последовательность в порции заданного размера (или меньше, если элементов не хватает):

```
foreach (int[] chunk in new[] { 1, 2, 3, 4, 5, 6, 7, 8 }.Chunk (3))
    Console.WriteLine (string.Join (", ", chunk));
```

Вывод:

```
1, 2, 3
4, 5, 6
7, 8
```

Операции над множествами

`IEnumerable<TSource>, IEnumerable<TSource> → IEnumerable<TSource>`

Метод	Описание	Эквиваленты в SQL
<code>Concat</code>	Возвращает результат конкатенации элементов в каждой из двух последовательностей	<code>UNION ALL</code>
<code>Union,</code> <code>UnionBy</code>	Возвращают результат конкатенации элементов в каждой из двух последовательностей, исключая дубликаты	<code>UNION</code>
<code>Intersect,</code> <code>IntersectBy</code>	Возвращают элементы, присутствующие в обеих последовательностях	<code>WHERE ... IN (...)</code>
<code>Except,</code> <code>ExceptBy</code>	Возвращают элементы, присутствующие в первой, но не во второй последовательности	<code>EXCEPT или WHERE... NOT IN (...)</code>

Concat, Union, UnionBy

Операция `Concat` возвращает все элементы из первой последовательности, за которыми следуют все элементы из второй последовательности. Операция `Union` делает то же самое, но удаляет любые дубликаты:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IQueryable<int>
    concat = seq1.Concat (seq2),           // { 1, 2, 3, 3, 4, 5 }
    union = seq1.Union (seq2);           // { 1, 2, 3, 4, 5 }
```

Явное указание аргумента типа удобно, когда последовательности типизированы по-разному, но элементы имеют общий базовый тип. Например, благодаря API-интерфейсу рефлексии (глава 18) методы и свойства представлены с помощью классов `MethodInfo` и `PropertyInfo`, которые имеют общий базовый класс по имени `MemberInfo`. Мы можем выполнить конкатенацию методов и свойств, явно указывая базовый класс при вызове `Concat`:

```
MethodInfo[] methods = typeof (string).GetMethods();
 PropertyInfo[] props = typeof (string).GetProperties();
 I Enumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

В следующем примере мы фильтруем методы перед конкатенацией:

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MethodInfo> (props);
```

Данный пример полагается на варианность параметров типа в интерфейсе: `methods` относится к типу `IEnumerable<MethodInfo>`, который требует ковариантного преобразования в тип `IEnumerable<MemberInfo>`. Пример является хорошей иллюстрацией того, как варианность делает поведение типов более похожим на то, что ожидается.

Метод `UnionBy` (появившийся в .NET 6) принимает селектор ключей, который используется для определения того, является ли элемент дубликатом. В следующем примере выполняется объединение, нечувствительное к регистру символов:

```
string[] seq1 = { "A", "b", "C" };
string[] seq2 = { "a", "B", "c" };
var union = seq1.UnionBy (seq2, x => x.ToUpperInvariant ());
// объединение: { "A", "b", "C" }
```

В данном случае того же самого результата можно добиться с помощью метода `Union`, если предоставить ему компаратор эквивалентности:

```
var union = seq1.Union (seq2, StringComparer.InvariantCultureIgnoreCase);
```

Intersect, IntersectBy, Except, ExceptBy

Операция `Intersect` возвращает элементы, имеющиеся в двух последовательностях. Операция `Except` возвращает элементы из первой последовательности, которые *не* присутствуют во второй последовательности:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
commonality = seq1.Intersect (seq2),           // { 3 }
difference1 = seq1.Except    (seq2),           // { 1, 2 }
difference2 = seq2.Except    (seq1);           // { 4, 5 }
```

Внутренне метод `Enumerable.Except` загружает все элементы первой последовательности в словарь и затем удаляет из словаря элементы, присутствующие во второй последовательности. Эквивалентом такой операции в SQL является подзапрос `NOT EXISTS` или `NOT IN`:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

Методы `IntersectBy` и `ExceptBy` (появившиеся в .NET 6) позволяют указывать селектор ключей, которые применяется перед выполнением сравнения эквивалентности (см. обсуждение `UnionBy` в предыдущем разделе).

Методы преобразования

Язык LINQ в основном имеет дело с последовательностями — другими словами, с коллекциями типа `IEnumerable<T>`. Методы преобразования преобразуют коллекции `IEnumerable<T>` в коллекции других типов и наоборот.

Метод	Описание
OfType	Преобразует IEnumerable в IEnumerable<T>, отбрасывая некорректно типизированные элементы
Cast	Преобразует IEnumerable в IEnumerable<T>, генерируя исключение при наличии некорректно типизированных элементов
ToArray	Преобразует IEnumerable<T> в T[]
ToList	Преобразует IEnumerable<T> в List<T>
ToDictionary	Преобразует IEnumerable<T> в Dictionary<TKey, TValue>
ToLookup	Преобразует IEnumerable<T> в ILookup<TKey, TElement>
AsEnumerable	Приводит вверх к IEnumerable<T>
AsQueryable	Приводит или преобразует в IQueryable<T>

OfType и Cast

Методы OfType и Cast принимают необобщенную коллекцию IEnumerable и выдают обобщенную последовательность IEnumerable<T>, которую впоследствии можно запрашивать:

```
ArrayList classicList = new ArrayList(); // из System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Когда методы Cast и OfType встречают входной элемент, который имеет несовместимый тип, их поведение отличается. Метод Cast генерирует исключение, а OfType игнорирует такой элемент. Продолжим предыдущий пример:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
sequence2 = classicList.OfType<int>(), // Нормально - проблемный
// элемент DateTime игнорируется
sequence3 = classicList.Cast<int>(); // Генерируется исключение
```

Правила совместимости элементов точно соответствуют правилам для операции is в языке C# и, следовательно, предусматривают только ссылочные и распаковывающие преобразования. Мы можем увидеть это, взглянув на внутреннюю реализацию OfType:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
    foreach (object element in source)
        if (element is TSource)
            yield return (TSource)element;
}
```

Метод Cast имеет идентичную реализацию за исключением того, что опускает проверку на предмет совместимости типов:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
    foreach (object element in source)
        yield return (TSource)element;
}
```

Из реализаций следует, что использовать Cast для выполнения числовых или специальных преобразований нельзя (взамен для них придется выполнять операцию Select). Другими словами, операция Cast не настолько гибкая, как операция приведения C#:

```
int i = 3;
long l = i;           // Неявное числовое преобразование int в long
int i2 = (int) l;    // Явное числовое преобразование long в int
```

Продемонстрировать сказанное можно, попробовав применить операции OfType или Cast для преобразования последовательности значений int в последовательность значений long:

```
int[] integers = { 1, 2, 3 };
IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

При перечислении test1 выдает ноль элементов, а test2 генерирует исключение. Просмотр реализации метода OfType четко проясняет причину. После подстановки TSource мы получаем следующее выражение:

```
(element is long)
```

которое возвращает false, когда element имеет тип int, из-за отсутствия отношения наследования.



Причина того, что test2 генерирует исключение при перечислении, несколько тоньше. В реализации метода Cast обратите внимание на то, что element имеет тип object. Когда TSource является типом значения, среда CLR предполагает, что это *распаковывающее преобразование*, и синтезирует метод, который воспроизводит сценарий, описанный в разделе “Упаковка и распаковка” главы 3:

```
int value = 123;
object element = value;
long result = (long) element; // Генерируется исключение
```

Поскольку переменная element объявлена с типом object, выполняется приведение object к long (распаковка), а не числовое преобразование int в long. Распаковывающие операции требуют точного соответствия типов, поэтому распаковка object в long терпит неудачу, когда элемент является int.

Как предполагалось ранее, решение предусматривает использование обычной операции Select:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

Операции OfType и Cast также удобны для приведения вниз элементов в обобщенной входной коллекции. Например, если имеется входная коллекция типа `IEnumerable<Fruit>` (фрукты), то `OfType<Apple>` (яблоки) возвратит только яблоки. Это особенно полезно в LINQ to XML (глава 10).

Операция Cast поддерживается в синтаксисе запросов: понадобится лишь предварить переменную диапазона нужным типом:

```
from TreeNode node in myTreeView.Nodes
```

```
...
```

ToDictionary, ToList, ToDictionary, ToHashSet, ToLookup

Операции `ToDictionary`, `ToList` и `ToHashSet` выдают результаты в массив, `List<T>` или `HashSet<T>`. При выполнении они вызывают немедленное перечисление входной последовательности. За примерами обращайтесь в раздел “Отложенное выполнение” главы 8.

Операции `ToDictionary` и `ToLookup` принимают описанные ниже аргументы.

Аргумент	Тип
Входная последовательность	<code>IEnumerable<TSource></code>
Селектор ключей	<code>TSource => TKey</code>
Селектор элементов (необязательный)	<code>TSource => TElement</code>
Компаратор (необязательный)	<code>IEqualityComparer<TKey></code>

Операция `ToDictionary` также приводит к немедленному перечислению последовательности с записью результатов в обобщенный словарь `Dictionary`. Представляемое выражение селектора ключей (`keySelector`) должно вычисляться как уникальное значение для каждого элемента во входной последовательности; в противном случае генерируется исключение. Напротив, операция `ToLookup` позволяет множеству элементов иметь один и тот же ключ. Объекты `Lookup` рассматривались в разделе “Выполнение соединений с помощью объектов `Lookup`” ранее в главе.

AsEnumerable и AsQueryable

Операция `AsEnumerable` выполняет приведение вверх последовательности к типу `IEnumerable<T>`, заставляя компилятор привязывать последующие операции запросов к методам из класса `Enumerable`, а не `Queryable`. За примером обращайтесь в раздел “Комбинирование интерпретируемых и локальных запросов” главы 8.

Операция `AsQueryable` выполняет приведение вниз последовательности к типу `IQueryable<T>`, если последовательность реализует этот интерфейс. В противном случае операция создает оболочку `IQueryable<T>` вокруг локального запроса.

Операции над элементами

IEnumerable<TSource> → TSource

Метод	Описание	Эквиваленты в SQL
First, FirstOrDefault	Возвращают первый элемент в последовательности, необязательно удовлетворяющий предикату	SELECT TOP 1... ORDER BY...
Last, LastOrDefault	Возвращают последний элемент в последовательности, необязательно удовлетворяющий предикату	SELECT TOP 1... ORDER BY...DESC
Single, SingleOrDefault	Эквивалентны операциям First/ FirstOrDefault, но генерируют исключение, если обнаружено более одного совпадения	
ElementAt, ElementAtOrDefault	Возвращают элемент в указанной позиции	Генерируется исключение
MinBy, MaxBy	Возвращают элемент с наименьшим или наибольшим значением	Генерируется исключение
DefaultIfEmpty	Возвращает одноИлементную последовательность, значением которой является default(TSource), если последовательность не содержит элементов	OUTER JOIN

Методы с именами, завершающимися на OrDefault, вместо генерации исключения возвращают default(TSource), если входная последовательность является пустой или отсутствуют элементы соответствующие заданному предикату.

Значение default(TSource) равно null для элементов ссылочных типов, false для элементов типа bool и 0 для элементов числовых типов.

First, Last, Single

Аргумент	Тип
Входная последовательность	IEnumerable<TSource>
Предикат (необязательный)	TSource => bool

В следующем примере демонстрируется работа операций First и Last:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int first      = numbers.First();                      // 1
int last       = numbers.Last();                       // 5
int firstEven  = numbers.First(n => n % 2 == 0);     // 2
int lastEven   = numbers.Last(n => n % 2 == 0);      // 4
```

Ниже проиллюстрирована работа операций First и FirstOrDefault:

```
int firstBigError = numbers.First (n => n > 10); // Генерируется исключение
int firstBigNumber = numbers.FirstOrDefault (n => n > 10); // 0
```

Чтобы не генерировалось исключение, операция Single требует наличия в точности одного совпадающего элемента, а SingleOrDefault — одного или нуля совпадающих элементов:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0); // 3
int divBy2Err = numbers.Single (n => n % 2 == 0); // Ошибка: совпадение
// дают 2 и 4
int singleError = numbers.Single (n => n > 10); // Ошибка
int noMatches = numbers.SingleOrDefault (n => n > 10); // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0); // Ошибка
```

В данном семействе операций над элементами Single является самой “придирчивой”. С другой стороны, операции FirstOrDefault и LastOrDefault наиболее толерантны.

В EF Core операция Single часто используется для извлечения строки из таблицы по первичному ключу:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

ElementAt

Аргумент	Тип
Входная последовательность	IEnumerable<TSource>
Индекс элемента для возврата	int

Операция ElementAt извлекает *n*-ный элемент из последовательности:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int third = numbers.ElementAt (2); // 3
int tenthError = numbers.ElementAt (9); // Генерируется исключение
int tenth = numbers.ElementAtOrDefault (9); // 0
```

Метод Enumerable.ElementAt написан так, что если входная последовательность реализует интерфейс IList<T>, тогда он вызывает индексатор, определенный в IList<T>. В противном случае он выполняет перечисление *n* раз и затем возвращает следующий элемент. Операция ElementAt в EF Core не поддерживается.

MinBy и MaxBy

Операции MinBy и MaxBy (появившиеся в .NET 6) возвращают элемент с наименьшим или наибольшим значением, как определено селектором ключей:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
Console.WriteLine (names.MaxBy (n => n.Length)); // Harry
```

Напротив, операции Min и Max (которые рассматриваются в следующем разделе) сами возвращают наименьшее или наибольшее значение:

```
Console.WriteLine (names.Max (n => n.Length)); // 5
```

Если два или большее количество элементов имеют одинаковое минимальное или максимальное значение, тогда MinBy или MaxBy возвращает первое:

```
Console.WriteLine (names.MinBy (n => n.Length)); // Tom
```

Если входная последовательность пуста, то операции MinBy и MaxBy возвращают null, если тип элемента допускает значение null (или генерирует исключение, если тип элемента не допускает null).

DefaultIfEmpty

Операция DefaultIfEmpty возвращает последовательность с единственным элементом, значением которого будет default(TSource), если входная последовательность не содержит элементов. В противном случае она возвращает неизмененную входную последовательность. Операция DefaultIfEmpty применяется при написании плоских внутренних соединений: за деталями обращайтесь в разделы “Выполнение внешних соединений с помощью SelectMany” и “Плоские внешние соединения” ранее в главе.

Методы агрегирования

IEnumerable<TSource> → скаляр

Метод	Описание	Эквиваленты в SQL
Count, LongCount	Возвращают количество элементов во входной последовательности, необязательно удовлетворяющих предикату	COUNT (...)
Min, Max	Возвращают наименьший или наибольший элемент в последовательности	MIN (...), MAX (...)
Sum, Average	Подсчитывают числовую сумму или среднее значение для элементов в последовательности	SUM (...), AVG (...)
Aggregate	Выполняет специальное агрегирование	Генерируется исключение

Count и LongCount

Аргумент	Тип
Входная последовательность	IEnumerable<TSource>
Предикат (необязательный)	TSource => bool

Операция Count просто выполняет перечисление последовательности, возвращая количество элементов:

```
int fullCount = new int[] { 5, 6, 7 }.Count(); // 3
```

Внутренняя реализация метода Enumerable.Count проверяет входную последовательность на предмет реализации ею интерфейса ICollection<T>. Если он реализован, тогда просто производится обращение к свойству

`ICollection<T>.Count`. В противном случае осуществляется перечисление по элементам последовательности с инкрементированием счетчика.

Можно дополнительно указать предикат:

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c)); // 3
```

Операция `LongCount` делает ту же работу, что и `Count`, но возвращает 64-битное целочисленное значение, позволяя последовательностям содержать более двух миллиардов элементов.

Min И Max

Аргумент	Тип
Входная последовательность	<code>IEnumerable<TSource></code>
Селектор результатов (необязательный)	<code>TSource => TResult</code>

Операции `Min` и `Max` возвращают наименьший или наибольший элемент из последовательности:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min(); // 14;
int largest = numbers.Max(); // 32;
```

Если указано выражение селектора, тогда каждый элемент сначала проецируется:

```
int smallest = numbers.Max (n => n % 10); // 8;
```

Выражение селектора будет обязательным, если элементы сами по себе не являются внутренне сопоставимыми — другими словами, если они не реализуют интерфейс `IComparable<T>`:

```
Purchase runtimeError = dbContext.Purchases.Min (); // Ошибка
decimal? lowestPrice = dbContext.Purchases.Min (p => p.Price); // Нормально
```

Выражение селектора определяет не только то, как элементы сравниваются, но также и финальный результат. В предыдущем примере финальным результатом оказывается десятичное значение, а не объект покупки. Для получения самой дешевой покупки понадобится подзапрос:

```
Purchase cheapest = dbContext.Purchases
    .Where (p => p.Price == dbContext.Purchases.Min (p2 => p2.Price))
    .FirstOrDefault();
```

В данном случае можно было бы сформулировать запрос без агрегации за счет использования операции `OrderBy` и затем `FirstOrDefault`.

Sum И Average

Аргумент	Тип
Входная последовательность	<code>IEnumerable<TSource></code>
Селектор результатов (необязательный)	<code>TSource => TResult</code>

`Sum` и `Average` представляют собой операции агрегирования, которые применяются подобно `Min` и `Max`:

```
decimal[] numbers = { 3, 4, 8 };
decimal sumTotal = numbers.Sum(); // 15
decimal average = numbers.Average(); // 5 (среднее значение)
```

Следующий запрос возвращает общую длину всех строк в массиве `names`:

```
int combinedLength = names.Sum(s => s.Length); // 19
```

Операции `Sum` и `Average` довольно ограничены в своей типизации. Их определения жестко привязаны к каждому числовому типу (`int`, `long`, `float`, `double`, `decimal`, а также версии этих типов, допускающие `null`). Напротив, операции `Min` и `Max` могут напрямую оперировать на всех типах, которые реализуют интерфейс `IComparable<T>` — например, `string`.

Более того, операция `Average` всегда возвращает либо тип `decimal`, либо тип `double` в соответствии со следующей таблицей.

Тип селектора	Тип результата
<code>decimal</code>	<code>decimal</code>
<code>float</code>	<code>float</code>
<code>int, long, double</code>	<code>double</code>

Это значит, что показанный ниже код не скомпилируется (будет выдано сообщение о невозможности преобразования `double` в `int`):

```
int avg = new int[] { 3, 4 }.Average();
```

Но следующий код скомпилируется:

```
double avg = new int[] { 3, 4 }.Average(); // 3.5
```

Операция `Average` неявно повышает входные значения, чтобы избежать потери точности. Выше в примере мы усредняем целочисленные значения и получаем 3.5 без необходимости в приведении входного элемента:

```
double avg = numbers.Average(n => (double) n);
```

При запрашивании базы данных операции `Sum` и `Average` транслируются в стандартные агрегации SQL. Представленный далее запрос возвращает заказчиков, у которых средняя покупка превышает сумму \$500:

```
from c in dbContext.Customers
where c.Purchases.Average(p => p.Price) > 500
select c.Name;
```

Aggregate

Операция `Aggregate` позволяет указывать специальный алгоритм накопления для реализации необычных агрегаций. Операция `Aggregate` в EF Core не поддерживается, а сценарии ее использования стоят несколько особняком. Ниже показано, как с помощью `Aggregate` выполнить работу операции `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n); // 6
```

Первым аргументом операции `Aggregate` является *начальное значение*, с которого стартует накопление. Второй аргумент — выражение для обновления накопленного значения заданным новым элементом. Можно дополнительно предоставить третий аргумент, предназначенный для проецирования финального результирующего значения из накопленного значения.



Большинство задач, для которых была спроектирована операция `Aggregate`, можно легко решить с помощью цикла `foreach` — к тому же с применением более знакомого синтаксиса. Преимущество использования `Aggregate` заключается в том, что построение крупных или сложных агрегаций может быть автоматически распараллелено посредством PLINQ (см. главу 22).

Агрегации без начального значения

При вызове операции `Aggregate` начальное значение может быть опущено; тогда первый элемент становится *неявным начальным значением* и агрегация продолжается со второго элемента. Ниже приведен предыдущий пример без использования начального значения:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n); // 6
```

Он дает тот же результат, что и ранее, но здесь выполняется *другое вычисление*. В предшествующем примере мы вычисляли $0+1+2+3$, а теперь вычисляем $1+2+3$. Лучше проиллюстрировать отличие поможет указание умножения вместо сложения:

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n); // 0*1*2*3 = 0
int y = numbers.Aggregate ( (prod, n) => prod * n); // 1*2*3 = 6
```

Как будет показано в главе 22, агрегации без начального значения обладают преимуществом параллелизма, не требуя применения специальных перегруженных версий. Тем не менее, с такими агрегациями связан ряд проблем.

Проблемы с агрегациями без начального значения

Методы агрегации без начального значения рассчитаны на использование с делегатами, которые являются *коммутативными* и *ассоциативными*. В случае их применения по-другому результат будет либо *непонятным* (в обычных запросах), либо *недетерминированным* (при распараллеливании запроса с помощью PLINQ). Например, рассмотрим следующую функцию:

```
(total, n) => total + n * n
```

Она не коммутативна и не ассоциативна. (Скажем, $1 + 2 * 2 \neq 2 + 1 * 1$). Давайте посмотрим, что произойдет, если мы воспользуемся ею для суммирования квадратов чисел 2, 3 и 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n); // 27
```

Вместо вычисления:

```
2*2 + 3*3 + 4*4 // 29
```

она вычисляет вот что:

```
2 + 3*3 + 4*4 // 27
```

Исправить ее можно несколькими способами. Для начала мы могли бы включить 0 в качестве первого элемента:

```
int[] numbers = { 0, 2, 3, 4 };
```

Это не только лишено элегантности, но будет еще и давать некорректные результаты при распараллеливании, поскольку PLINQ рассчитывает на ассоциативность функции, выбирая несколько элементов в качестве начальных. Чтобы проиллюстрировать сказанное, определим функцию агрегирования следующим образом:

```
f(total, n) => total + n * n
```

Вот как ее вычислит инфраструктура LINQ to Objects:

```
f(f(f(0, 2), 3), 4)
```

В то же время PLINQ может делать такое:

```
f(f(0,2),f(3,4))
```

с приведенным ниже результатом:

Первая часть: $a = 0 + 2 \cdot 2 (= 4)$
Вторая часть: $b = 3 + 4 \cdot 4 (= 19)$
Финальный результат: $a + b \cdot b (= 365)$
или даже так: $b + a \cdot a (= 35)$

Существуют два надежных решения. Первое — превратить функцию в агрегацию с нулевым начальным значением. Единственная сложность в том, что для PLINQ потребовалось бы использовать специальную перегруженную версию функции, чтобы запрос не выполнялся последовательно (как объясняется в разделе “Оптимизация PLINQ” главы 22).

Второе решение предусматривает реструктуризацию запроса, так что функция агрегации становится коммутативной и ассоциативной:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```



Разумеется, в таких простых сценариях вы можете (и должны) применять операцию Sum вместо Aggregate:

```
int sum = numbers.Sum (n => n * n);
```

На самом деле с помощью только операций Sum и Average можно добиться довольно многоного. Например, Average можно использовать для вычисления среднеквадратического значения:

```
Math.Sqrt (numbers.Average (n => n * n))
```

и даже стандартного отклонения:

```

double mean = numbers.Average();
double sdev = Math.Sqrt (numbers.Average (n =>
{
    double dif = n - mean;
    return dif * dif;
}));
```

Оба примера безопасны, эффективны и поддаются распараллеливанию. В главе 22 мы приведем практический пример специальной агрегации, которая не может быть сведена к Sum или Average.

Квантификаторы

`IEnumerable<TSource> → значение bool`

Метод	Описание	Эквиваленты в SQL
Contains	Возвращает true, если входная последовательность содержит заданный элемент	WHERE ... IN (...)
Any	Возвращает true, если любой элемент удовлетворяет заданному предикату	WHERE ... IN (...)
All	Возвращает true, если все элементы удовлетворяют заданному предикату	WHERE (...)
SequenceEqual	Возвращает true, если вторая последовательность содержит элементы, идентичные элементам в первой последовательности	.

Contains и Any

Метод Contains принимает аргумент типа `TSource`, а Any — необязательный *предикат*.

Операция Contains возвращает true, если заданный элемент присутствует в последовательности:

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3); // true
```

Операция Any возвращает true, если указанное выражение дает значение true хотя бы для одного элемента. Предшествующий пример можно переписать с применением операции Any:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3); // true
```

Операция Any может делать все, что делает Contains, и даже больше:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10); // false
```

Вызов метода Any без предиката приводит к возвращению true, если последовательность содержит один или большее число элементов. Ниже показан другой способ записи предыдущего запроса:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

Операция Any особенно удобна в подзапросах и часто используется при за-рашивании баз данных, например:

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

All и SequenceEqual

Операция All возвращает true, если все элементы удовлетворяют предикату. Следующий запрос возвращает заказчиков с покупками на сумму меньше \$100:

```
dbContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

Операция SequenceEqual сравнивает две последовательности. Для возвращения true обе последовательности должны иметь идентичные элементы, расположенные в одинаковом порядке. Можно дополнительно указать компаратор эквивалентности; по умолчанию применяется EqualityComparer<T>.Default.

Методы генерации

Ничего на входе → IEnumerable<TResult>

Метод	Описание
Empty	Создает пустую последовательность
Repeat	Создает последовательность повторяющихся элементов
Range	Создает последовательность целочисленных значений

Empty, Repeat и Range являются статическими (не расширяющими) методами, которые создают простые локальные последовательности.

Empty

Метод Empty создает пустую последовательность и требует только аргумента типа:

```
foreach (string s in Enumerable.Empty<string>())
    Console.Write (s); // <ничего>
```

В сочетании с операцией ?? метод Empty выполняет действие, противоположное действию метода DefaultIfEmpty. Например, предположим, что имеется зубчатый массив целых чисел, и требуется получить все целые числа в виде единственного плоского списка. Следующий запрос SelectMany терпит неудачу, если любой из внутренних массивов зубчатого массива оказывается null:

```
int[][] numbers =
{
    new int[] { 1, 2, 3 },
    new int[] { 4, 5, 6 },
    null           // Это значение null приводит к отказу запроса
};
IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Проблема решается за счет использования комбинации метода `Empty` с операцией `??`:

```
IEnumerable<int> flat = numbers
    .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());
foreach (int i in flat)
    Console.Write (i + " ");      // 1 2 3 4 5 6
```

Range И Repeat

Метод `Range` принимает начальный индекс и счетчик (оба значения являются целочисленными):

```
foreach (int i in Enumerable.Range (5, 3))
    Console.Write (i + " ");      // 5 6 7
```

Метод `Repeat` принимает элемент, подлежащий повторению, и количество повторений:

```
foreach (bool x in Enumerable.Repeat (true, 3))
    Console.Write (x + " ");      // True True True
```



LINQ to XML

Платформа .NET предоставляет несколько API-интерфейсов для работы с XML-данными. Основным выбором для универсальной обработки XML-документов является инфраструктура LINQ to XML, которая состоит из легковесной, дружественной к LINQ объектной модели XML-документа и набора дополнительных операций запросов.

В настоящей главе мы сосредоточим внимание целиком на LINQ to XML. В главе 11 мы раскроем однородные средства чтения/записи XML, а в дополнительных материалах, доступных на веб-сайте издательства, обсудим типы для работы со схемами и таблицами стилей. Кроме того, платформа .NET включает унаследованную объектную модель документа (document object model — DOM), основанную на классе `XmlDocument`, которая здесь не рассматривается.



DOM-модель LINQ to XML исключительно хорошо спроектирована и отличается высокой производительностью. Даже без LINQ эта модель полезна в качестве легковесного фасада для низкоуровневых классов `XmlReader` и `XmlWriter`.

Все типы LINQ to XML определены в пространстве имен `System.Xml.Linq`.

Обзор архитектуры

Раздел начинается с очень краткого введения в концепции DOM-модели и продолжается объяснением логических обоснований, лежащих в основе DOM-модели LINQ to XML.

Что собой представляет DOM-модель

Рассмотрим следующее содержимое XML-файла:

```
<?xml version=1.0 encoding=utf-8?>
<customer id=123 status=archived>
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

Как и все XML-файлы, он начинается с объявления, после которого следует корневой элемент по имени `customer`. Элемент `customer` имеет два атрибута, с каждым из которых связано имя (`id` и `status`) и значение ("123" и "archived"). Внутри `customer` присутствуют два дочерних элемента, `firstname` и `lastname`, каждый из которых имеет простое текстовое содержимое ("Joe" и "Bloggs").

Каждая из упомянутых выше конструкций — объявление, элемент, атрибут, значение и текстовое содержимое — может быть представлена с помощью класса. А если такие классы имеют свойства коллекций для хранения дочернего содержимого, то для полного описания документа мы можем построить дерево объектов. Это и называется объектной моделью документа, или DOM-моделью.

DOM-модель LINQ to XML

Инфраструктура LINQ to XML состоит из двух частей:

- DOM-модель XML, которую мы называем X-DOM;
- набор из примерно десятка дополнительных операций запросов.

Как и можно было ожидать, модель X-DOM состоит из таких типов, как `XDocument`, `XElement` и `XAttribute`. Интересно отметить, что типы X-DOM не привязаны к LINQ — модель X-DOM можно загружать, создавать ее экземпляры, обновлять и сохранять вообще без написания каких-либо запросов LINQ.

И наоборот, LINQ можно использовать для выдачи запросов к DOM-модели, созданной старыми типами, совместимыми с W3C. Однако такой подход утомителен и обладает ограниченными возможностями. Отличительной особенностью модели X-DOM является дружественность к LINQ, что означает следующее:

- она имеет методы, выпускающие удобные последовательности `IEnumerable`, которые можно запрашивать;
- ее конструкторы спроектированы так, что дерево X-DOM можно построить посредством проецирования LINQ.

Обзор модели X-DOM

На рис. 10.1 показаны основные типы модели X-DOM. Самым часто применяемым типом является `XElement`. Тип `XObject` представляет собой корень иерархии наследования, а типы `XElement` и `XDocument` — корни иерархии включения. На рис. 10.2 изображено дерево X-DOM, созданное из приведенного ниже кода:

```
string xml = @<customer id='123' status='archived'>
    <firstname>Joe</firstname>
    <lastname>Bloggs<!--nice name--></lastname>
</customer>;
XElement customer = XElement.Parse (xml);
```

Тип `XObject` является абстрактным базовым классом для всего XML-содержимого. Он определяет ссылку на элемент `Parent` в контейнерном дереве, а также необязательный объект `XDocument`.

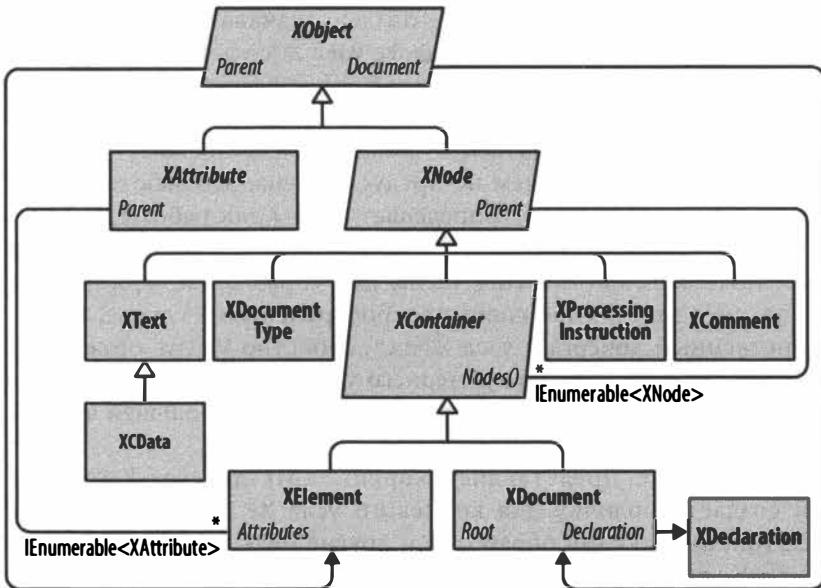


Рис. 10.1. Основные типы X-DOM

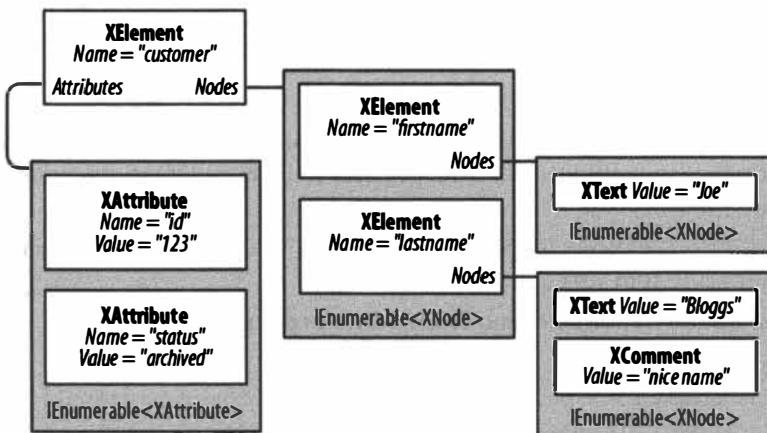


Рис. 10.2. Простое дерево X-DOM

Тип **XNode** — базовый класс для большей части XML-содержимого, исключая атрибуты. Отличительная особенность объекта **XNode** в том, что он может находиться в упорядоченной коллекции смешанных типов **XNode**. Например, взгляните на такой XML-код:

```

<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
  
```

Внутри родительского элемента `<data>` сначала определен узел `XText` (`Hello world`), затем узел `XElement`, далее узел `XComment` и, наконец, еще один узел `XElement`. Напротив, объект `XAttribute` будет допускать в качестве равноправных узлов только другие объекты `XAttribute`.

Хотя `XNode` может обращаться к своему родительскому узлу `XElement`, концепция дочерних узлов в нем не предусмотрена: это забота его подклассов `XContainer`. Класс `XContainer` определяет члены для работы с дочерними узлами и является абстрактным базовым классом для `XElement` и `XDocument`.

В классе `XElement` определены члены для управления атрибутами, а также члены `Name` и `Value`. В (довольно распространенном) случае, когда элемент имеет единственный дочерний узел `XText`, свойство `Value` объекта `XElement` инкапсулирует содержимое этого дочернего узла для операций `get` и `set`, устраняя излишнюю навигацию. Благодаря `Value` можно по большей части избежать прямого взаимодействия с узлами `XText`.

Класс `XDocument` представляет корень XML-дерева. Выражаясь более точно, он создает оболочку для корневого узла `XElement`, добавляя объект `XDeclaration`, инструкции обработки и другие мелкие детали корневого уровня. В отличие от DOM-модели W3C использовать класс `XDocument` необязательно: вы можете загружать, манипулировать и сохранять модель X-DOM, даже не создавая объект `XDocument`! Необязательность `XDocument` также означает возможность эффективного и легкого перемещения поддерева узла в другую иерархию X-DOM.

Загрузка и разбор

Классы `XElement` и `XDocument` предоставляют статические методы `Load` и `Parse`, предназначенные для построения дерева X-DOM из существующего источника:

- метод `Load` строит дерево X-DOM из файла, URI, объекта `Stream`, `TextReader` или `XmlReader`;
- метод `Parse` строит дерево X-DOM из строки.

Например:

```
XDocument fromWeb = XDocument.Load ("http://albahari.com/sample.xml");
 XElement fromFile = XElement.Load (@e:\media\somefile.xml);
 XElement config = XElement.Parse (
 @"<configuration>
    <client enabled='true'>
        <timeout>30</timeout>
    </client>
</configuration>");
```

В последующих разделах мы покажем, каким образом выполнять обход и обновление дерева X-DOM. В качестве краткого обзора взгляните, как манипулировать только что наполненным элементом `config`:

```
foreach ( XElement child in config.Elements())
    Console.WriteLine (child.Name); // client
 XElement client = config.Element ("client");
```

```
bool enabled = (bool) client.Attribute ("enabled"); // Прочитать атрибут
Console.WriteLine (enabled); // True
client.Attribute ("enabled").SetValue (!enabled); // Обновить атрибут
int timeout = (int) client.Element ("timeout"); // Прочитать элемент
Console.WriteLine (timeout); // 30
client.Element ("timeout").SetValue (timeout * 2); // Обновить элемент
client.Add (new XElement ("retries", 3)); // Добавить новый элемент
Console.WriteLine (config); // Неявно вызвать метод config.ToString
```

Ниже показан результат последнего вызова `Console.WriteLine`:

```
<configuration>
  <client enabled=false>
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration>
```



Класс `XNode` также предоставляет статический метод `ReadFrom`, который создает экземпляр любого типа узла и наполняет его из `XmlReader`. В отличие от `Load` он останавливается после чтения одного (полного) узла, так что затем можно вручную продолжить чтение из `XmlReader`.

Можно также делать обратное действие и применять `XmlReader` или `XmlWriter` для чтения или записи `XNode` через методы `CreateReader` и `CreateWriter`.

Мы опишем средства чтения и записи XML и объясним, как ими пользоваться, в главе 11.

Сохранение и сериализация

Вызов метода `ToString` на любом узле преобразует его содержимое в XML-строку, сформированную с разрывами и отступами, как только что было показано. (Разрывы строки и отступы можно запретить, указав `SaveOptions.DisableFormatting` при вызове `ToString`.)

Классы `XElement` и `XDocument` также предлагают метод `Save`, который записывает модель X-DOM в файл, объект `Stream`, `TextWriter` или `XmlWriter`. Если указан файл, то автоматически записывается и XML-объявление. В классе `XNode` также определен метод `WriteTo`, который принимает объект `XmlWriter`.

Мы более подробно опишем обработку XML-объявлений при сохранении в разделе “Документы и объявления” далее в главе.

Создание экземпляра X-DOM

Вместо применения метода `Load` или `Parse` дерево X-DOM можно построить, вручную создавая объекты и добавляя их к родительскому узлу посредством метода `Add` класса `XContainer`.

Чтобы сконструировать объект `XElement` и `XAttribute`, нужно просто предоставить имя и значение:

```
XElement lastName = new XElement ("lastname", "Bloggs");
lastName.Add (new XComment ("nice name"));
XElement customer = new XElement ("customer");
customer.Add (new XAttribute ("id", 123));
customer.Add (new XElement ("firstname", "Joe"));
customer.Add (lastName);
Console.WriteLine (customer.ToString());
```

Вот результат:

```
<customer id=123>
<firstname>Joe</firstname>
<lastname>Bloggs<!--nice name--></lastname>
</customer>
```

При конструировании объекта XElement указывать значение необязательно — можно задать только имя элемента, а содержимое добавить позже. Обратите внимание, что когда предоставляется значение, простой строки вполне достаточно — в явном создании и добавлении дочернего узла XText нет необходимости. Модель X-DOM выполняет эту работу автоматически, так что приходится иметь дело только со значениями.

Функциональное построение

В предыдущем примере получить представление об XML-структуре на основании кода довольно-таки нелегко. Модель X-DOM поддерживает другой режим создания объектов, который называется функциональным построением (понятие, взятое из функционального программирования). При функциональном построении в единственном выражении строится целое дерево:

```
XElement customer =
    new XElement ("customer", new XAttribute ("id", 123),
        new XElement ("firstname", "joe"),
        new XElement ("lastname", "bloggs",
            new XComment ("nice name"))
    );
};
```

Такой подход обладает двумя преимуществами. Во-первых, код имеет сходство по форме с результирующим кодом XML. Во-вторых, он может быть включен в конструкцию select запроса LINQ. Например, следующий запрос выполняет проекцию из сущностного класса EF Core в модель X-DOM:

```
XElement query =
    new XElement ("customers",
        from c in dbContext.Customers.AsEnumerable()
        select
            new XElement ("customer", new XAttribute ("id", c.ID),
                new XElement ("firstname", c.FirstName),
                new XElement ("lastname", c.LastName,
                    new XComment ("nice name"))
            )
    );
};
```

Более подробно об этом пойдет речь в разделе “Проектирование в модель X-DOM” далее в главе.

Указание содержимого

Функциональное построение возможно из-за того, что конструкторы для XElement (и XDocument) перегружены с целью принятия массива типа object[] по имени params:

```
public XElement(XName name, params object[] content)
```

То же самое справедливо и в отношении метода Add в классе XContainer:

```
public void Add(params object[] content)
```

Таким образом, при построении или дополнении дерева X-DOM можно указывать любое количество дочерних объектов любых типов. Это работает, т.к. законным содержимым считается все, что угодно. Чтобы удостовериться в сказанном, необходимо посмотреть, как внутренне обрабатывается каждый объект содержимого. Ниже перечислены решения, которые по очереди принимает XContainer.

1. Если объект является null, то он игнорируется.
2. Если объект основан на XNode или XStreamingElement, тогда он добавляется в коллекцию Nodes в том виде, как есть.
3. Если объект является XAttribute, то он добавляется в коллекцию Attributes.
4. Если объект является строкой, тогда он помещается в узел XText и добавляется в коллекцию Nodes¹.
5. Если объект реализует интерфейс IEnumerable, то производится его перечисление с применением к каждому элементу тех же самых правил.
6. В противном случае объект преобразуется в строку, помещается в узел XText и затем добавляется в коллекцию Nodes².

В итоге все объекты попадают в одну из двух коллекций: Nodes или Attributes. Более того, любой объект является допустимым содержимым, потому что в конечном итоге на нем всегда можно вызывать метод ToString и трактовать его как узел XText.



Перед вызовом метода ToString на произвольном типе реализация XContainer сначала проверяет, не относится ли он к одному из следующих типов:

```
float, double, decimal, bool,  
DateTime, DateTimeOffset, TimeSpan
```

Если относится, тогда XContainer вызывает надлежащим образом типизированный метод ToString на вспомогательном классе XmlConvert вместо вызова ToString на самом объекте. Это гарантирует, что данные поддерживают обмен и совместимы со стандартными правилами форматирования XML.

¹ В действительности модель X-DOM внутренне оптимизирует данный шаг, храня простое текстовое содержимое в строке. Узел XText фактически не создается вплоть до вызова метода Nodes на XContainer.

² См. сноску 1.

Автоматическое глубокое копирование

Когда к элементу добавляется узел или атрибут (с помощью функционального построения либо посредством метода Add), ссылка на данный элемент присваивается свойству Parent добавляемого узла или атрибута. Узел может иметь только один родительский элемент: если вы добавляете узел, уже имеющий родительский элемент, ко второму родительскому элементу, то этот узел автоматически подвергается глубокому копированию. В следующем примере каждый заказчик имеет отдельную копию address:

```
var address = new XElement ("address",
    new XElement ("street", "Lawley St"),
    new XElement ("town", "North Beach")
);
var customer1 = new XElement ("customer1", address);
var customer2 = new XElement ("customer2", address);
customer1.Element ("address").Element ("street").Value = "Another St";
Console.WriteLine (
    customer2.Element ("address").Element ("street").Value); // Lawley St
```

Такое автоматическое дублирование сохраняет создание объектов модели X-DOM свободным от побочных эффектов — еще один признак функционального программирования.

Навигация и запросы

Как и можно было ожидать, в классах XNode и XContainer определены методы и свойства, предназначенные для обхода дерева X-DOM. Тем не менее, в отличие от обычной модели DOM такие методы и свойства не возвращают коллекцию, которая реализует интерфейс `IList<T>`. Взамен они возвращают либо одиночное значение, либо последовательность, реализующую интерфейс `IEnumerable<T>`, в отношении которой затем планируется выполнить запрос LINQ (или провести перечисление с помощью `foreach`). В результате появляется возможность запускать сложные запросы, а также решать простые задачи навигации с использованием знакомого синтаксиса запросов LINQ.



Имена элементов и атрибутов в X-DOM чувствительны к регистру — точно как в языке XML.

Навигация по дочерним узлам



Функции, помеченные звездочкой (*) в третьей колонке таблицы, также оперируют на *последовательностях* того же самого типа. Например, метод `Nodes` можно вызывать либо на объекте `XContainer`, либо на последовательности объектов `XContainer`. Такая возможность доступна благодаря расширяющим методам, которые определены в пространстве имен `System.Xml.Linq` — дополнительным операциям запросов, упомянутым в начале главы.

Возвращаемый тип	Члены	С чем работают
XNode	FirstNode { get; } LastNode { get; }	XContainer XContainer
IEnumerable<XNode>	Nodes() DescendantNodes() DescendantNodesAndSelf()	XContainer* XContainer* XElement*
XElement	Element(XName)	XContainer
IEnumerable< XElement >	Elements() Elements(XName) Descendants() Descendants(XName) DescendantsAndSelf() DescendantsAndSelf(XName)	XContainer* XContainer* XContainer* XContainer* XElement* XElement*
bool	HasElements { get; }	XElement

FirstNode, LastNode, Nodes

Свойства FirstNode и LastNode предоставляют прямой доступ к первому и последнему дочернему узлу; метод Nodes возвращает все дочерние узлы в виде последовательности. Все три функции принимают во внимание только непосредственных потомков. Например:

```
var bench = new XElement ("bench",
    new XElement ("toolbox",
        new XElement ("handtool", "Hammer"),
        new XElement ("handtool", "Rasp")
    ),
    new XElement ("toolbox",
        new XElement ("handtool", "Saw"),
        new XElement ("powertool", "Nailgun")
    ),
    new XComment ("Be careful with the nailgun")
);
foreach (XNode node in bench.Nodes())
    Console.WriteLine (node.ToString (SaveOptions.DisableFormatting) + ".");
```

Ниже показан вывод:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>.
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>.
<!--Be careful with the nailgun-->.
```

Извлечение элементов

Метод Elements возвращает только дочерние узлы типа XElement:

```
foreach ( XElement e in bench.Elements())
    Console.WriteLine (e.Name + "=" + e.Value); // toolbox=HammerRasp
                                                // toolbox=SawNailgun
```

Следующий запрос LINQ находит ящик с пневматическим молотком (nail gun):

```
IEnumerable<string> query =
    from toolbox in bench.Elements()
    where toolbox.Elements().Any (tool => tool.Value == "Nailgun")
    select toolbox.Value;
```

Вот результат:

```
{ "SawNailgun" }
```

В приведенном далее примере запрос SelectMany применяется для извлечения ручных инструментов (hand tool) из всех ящиков:

```
IEnumerable<string> query =
    from toolbox in bench.Elements()
    from tool in toolbox.Elements()
    where tool.Name == "handtool"
    select tool.Value;
```

Вот результат:

```
{ "Hammer", "Rasp", "Saw" }
```



Сам по себе метод Elements является эквивалентом запроса LINQ в отношении Nodes. Предыдущий запрос можно было бы начать следующим образом:

```
from toolbox in bench.Nodes().OfType< XElement >()
where ...
```

Метод Elements может также возвращать только элементы с заданным именем:

```
int x = bench.Elements ("toolbox").Count(); // 2
```

Данный код эквивалентен такому коду:

```
int x = bench.Elements().Where (e => e.Name == "toolbox").Count(); // 2
```

Кроме того, Elements определен как расширяющий метод, который принимает реализацию интерфейса IEnumerable<XContainer> или точнее аргумент следующего типа:

```
IEnumerable<T> where T : XContainer
```

Это позволяет ему работать также и с последовательностями элементов. С использованием метода Elements запрос, который ищет ручные инструменты во всех ящиках, можно записать так:

```
from tool in bench.Elements ("toolbox").Elements ("handtool")
select tool.Value;
```

Первый вызов Elements привязывается к методу экземпляра XContainer, а второй вызов Elements — к расширяющему методу.

Извлечение одиночного элемента

Метод Element (с именем в форме единственного числа) возвращает первый совпадающий элемент с заданным именем. Метод Element удобен для простой навигации вроде продемонстрированной ниже:

```
XElement settings = XElement.Load ("databaseSettings.xml");
string cx = settings.Element ("database").Element ("connectString").Value;
```

Вызов `Element` эквивалентен вызову метода `Elements` с последующим применением операции запроса `FirstOrDefault` языка LINQ с предикатом сопоставления по имени. Метод `Element` возвращает `null`, если запрошенный элемент не существует.



Вызов `Element("xyz").Value` сгенерирует исключение `NullReferenceException`, когда элемент `xyz` не существует. Если вместо исключения предпочтительнее получить значение `null`, тогда вместо обращения к свойству `Value` необходимо либо использовать null-условную операцию (`Element("xyz")?.Value`), либо привести `XElement` к типу `string`. Другими словами:

```
string xyz = (string) settings.Element ("xyz");
```

Прием работает из-за того, что в классе `XElement` определено явное преобразование в `string`, предназначенное как раз для такой цели!

Извлечение потомков

Класс `XContainer` также предлагает методы `Descendants` и `DescendantNodes`, которые возвращают дочерние элементы либо узлы вместе со всеми их дочерними элементами и т.д. (целое дерево). Метод `Descendants` принимает необязательное имя элемента. Возвращаясь к ранее рассмотренному примеру, вот как применить метод `Descendants` для поиска ручных инструментов:

```
Console.WriteLine (bench.Descendants ("handtool").Count()); // 3
```

Ниже продемонстрировано, что включаются и родительские, и листовые узлы:

```
foreach (XNode node in bench.DescendantNodes())
    Console.WriteLine (node.ToString (SaveOptions.DisableFormatting));
```

Вывод выглядит так:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

Следующий запрос извлекает из дерева X-DOM все комментарии, которые содержат слово “careful”:

```
IEnumerable<string> query =
    from c in bench.DescendantNodes().OfType<XComment>()
    where c.Value.Contains ("careful")
    orderby c.Value
    select c.Value;
```

Навигация по родительским узлам

Все классы XNode имеют свойство Parent и методы AncestorXXX, предназначенные для навигации по родительским узлам. Родительский узел всегда представляет собой объект XElement.

Возвращаемый тип	Члены	С чем работают
XElement	Parent { get; }	XNode
Enumerable<XElement>	Ancestors() Ancestors(XName) AncestorsAndSelf() AncestorsAndSelf(XName)	XNode XNode XElement XElement

Если x является XElement, тогда следующий код всегда выводит true:

```
foreach (XNode child in x.Nodes())
    Console.WriteLine (child.Parent == x);
```

Однако в случае, когда x представляет собой XDocument, все будет по-другому. Элемент XDocument особенный: он может иметь дочерние узлы, но никогда не может выступать родителем в отношении чего бы то ни было! Для доступа к XDocument должно использоваться свойство Document — оно работает на любом объекте в дереве X-DOM.

Метод Ancestors возвращает последовательность, первым элементом которой является Parent, следующим элементом — Parent.Parent и т.д. вплоть до корневого элемента.



Перейти к корневому элементу можно с помощью LINQ-запроса AncestorsAndSelf().Last().

Другой способ достигнуть того же результата предусматривает обращение к свойству Document.Root, хотя такой прием работает только при наличии XDocument.

Навигация по равноправным узлам

Возвращаемый тип	Члены	Определены в
bool	IsBefore(XNode node) IsAfter(XNode node)	XNode XNode
XNode	PreviousNode{ get; } NextNode{ get; }	XNode XNode
IEnumerable<XNode>	NodesBeforeSelf() NodesAfterSelf()	XNode XNode
IEnumerable< XElement >	ElementsBeforeSelf() ElementsBeforeSelf(XName name) ElementsAfterSelf() ElementsAfterSelf(XName name)	XNode XNode XNode XNode

С помощью свойств `PreviousNode` и `NextNode` (а также `FirstNode`/`LastNode`) узлы можно обходить с ощущением работы со связным списком. И это не случайно: внутренне узлы хранятся именно в связном списке.



Класс `XNode` внутренне применяет односвязный список, так что свойство `PreviousNode` не функционально.

Навигация по атрибутам

Возвращаемый тип	Члены	Определены в
<code>bool</code>	<code>HasAttributes { get; }</code>	<code>XElement</code>
<code>XAttribute</code>	<code>Attribute (XName name)</code> <code>FirstAttribute { get; }</code> <code>LastAttribute { get; }</code>	<code>XElement</code>
<code>IEnumerable<XAttribute></code>	<code>Attributes ()</code> <code>Attributes (XName name)</code>	<code>XElement</code>

В добавок в `XAttribute` определены свойства `PreviousAttribute` и `NextAttribute`, а также `Parent`.

Метод `Attributes`, который принимает имя, возвращает последовательность с нулем или одним элементом; в XML элемент не может иметь дублированные имена атрибутов.

Обновление модели X-DOM

Обновлять элементы и атрибуты можно следующими способами:

- вызвать метод `SetValue` или переустановить свойство `Value`;
- вызвать метод `SetElementValue` или `SetAttributeValue`;
- вызвать один из методов `RemoveXXX`;
- вызвать один из методов `AddXXX` или `ReplaceXXX`, указав новое содержимое.

Можно также переустанавливать свойство `Name` объектов `XElement`.

Обновление простых значений

Члены	С чем работают
<code>SetValue (object value)</code>	<code>XElement, XAttribute</code>
<code>Value { get; set }</code>	<code>XElement, XAttribute</code>

Метод `SetValue` заменяет содержимое элемента или атрибута простым значением. Установка свойства `Value` делает то же самое, но принимает только строковые данные. Мы подробно опишем эти функции в разделе “Работа со значениями” далее в главе.

Эффект от вызова метода `SetValue` (или переустановки свойства `Value`) заключается в замене всех дочерних узлов:

```
XElement settings = new XElement ("settings",
    new XElement ("timeout", 30)
);
settings.SetValue ("blah");
Console.WriteLine (settings.ToString()); // <settings>blah</settings>
```

Обновление дочерних узлов и атрибутов

Категория	Члены	С чем работают
Добавление	Add (params object[] content) AddFirst (params object[] content)	XContainer XContainer
Удаление	RemoveNodes() RemoveAttributes() RemoveAll()	XContainer XElement XElement
Обновление	ReplaceNodes (params object[] content) ReplaceAttributes (params object[] content) ReplaceAll (params object[] content) SetElementValue (XName name, object value) SetAttributeValue (XName name, object value)	XContainer XElement XElement XElement XElement

Наиболее удобными методами в данной группе являются последние два: `SetElementValue` и `SetAttributeValue`. Они служат сокращениями для создания экземпляра `XElement` либо `XAttribute` и затем его добавления посредством `Add` к родительскому узлу с заменой любого существующего элемента или атрибута, имеющего такое же имя:

```
XElement settings = new XElement ("settings");
settings.SetElementValue ("timeout", 30); // Добавляет дочерний узел
settings.SetElementValue ("timeout", 60); // Обновляет его значением 60
```

Метод `Add` добавляет дочерний узел к элементу или документу. Метод `AddFirst` делает то же самое, но вставляет узел в начало коллекции, а не в ее конец. С помощью метода `RemoveNodes` или `RemoveAttributes` можно удалить все дочерние узлы или атрибуты за один раз. Метод `RemoveAll` представляет собой эквивалент вызова обоих указанных методов.

Методы `ReplaceXXX` являются эквивалентами вызова сначала `RemoveXXX`, а затем `AddXXX`. Они получают копию входных данных, так что `e.ReplaceNodes (e.Nodes ())` работает ожидаемым образом.

Обновление через родительский элемент

Члены	С чем работают
AddBeforeSelf (params object[] content)	XNode
AddAfterSelf (params object[] content)	XNode
Remove()	XNode, XAttribute
ReplaceWith (params object[] content)	XNode

Методы AddBeforeSelf, AddAfterSelf, Remove и ReplaceWith не оперируют на дочерних узлах заданного узла. Взамен они работают с коллекцией, в которой находится сам узел. Это требует, чтобы узел имел родительский элемент — иначе генерируется исключение. Методы AddBeforeSelf и AddAfterSelf удобны для вставки узла в произвольную позицию:

```
XElement items = new XElement ("items",
    new XElement ("one"),
    new XElement ("three")
);
items.FirstNode.AddAfterSelf (new XElement ("two"));
```

Ниже показан результат:

```
<items><one /><two /><three /></items>
```

Вставка в произвольную позицию внутри длинной последовательности элементов на самом деле довольно эффективна, поскольку внутренне узлы хранятся в связном списке.

Метод Remove удаляет текущий узел из его родительского узла. Метод ReplaceWith делает то же самое, но затем вставляет в ту же самую позицию другое содержимое. Например:

```
XElement items = XElement.Parse ("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith (new XComment ("one was here"));
```

Вот результат:

```
<items><!--one was here--><two /><three /></items>
```

Удаление последовательности узлов или атрибутов

Благодаря расширяющим методам из пространства имен System.Xml.Linq метод Remove можно также вызывать на последовательности узлов или атрибутов. Взгляните на следующую модель X-DOM:

```
XElement contacts = XElement.Parse (
@<contacts>
<customer name='Mary' />
<customer name='Chris' archived='true' />
<supplier name='Susan'>
    <phone archived='true'>012345678<!--confidential--></phone>
</supplier>
</contacts>");
```

Приведенный ниже вызов удаляет всех заказчиков:

```
contacts.Elements ("customer").Remove();
```

Следующий оператор удаляет все архивные (“archived”) контакты (так что запись для Chris больше не будет видна):

```
contacts.Elements ().Where (e => (bool?) e.Attribute ("archived") == true)
    .Remove();
```

Если мы заменим вызов метода Elements вызовом Descendants, то все архивные элементы в DOM-модели больше не будут видны, и результат окажется следующим:

```
<contacts>
  <customer name=Mary />
  <supplier name=Susan />
</contacts>
```

В показанном ниже примере удаляются все контакты, которые имеют комментарий “*confidential*” (конфиденциально) в любом месте своего дерева:

```
contacts.Elements().Where (e => e.DescendantNodes()
                           .OfType<XComment>()
                           .Any (c => c.Value == "confidential")
                           ).Remove();
```

Результат будет таким:

```
<contacts>
  <customer name=Mary />
  <customer name=Chris archived=true />
</contacts>
```

Сравните это со следующим более простым запросом, который удаляет все узлы комментариев из дерева:

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```



Внутренне метод Remove сначала читает все совпадающие элементы во временный список, после чего организует его перечисление, чтобы выполнить удаление. Такой подход позволяет избежать ошибок, которые могут в противном случае возникнуть из-за удаления и запрашивания в один и тот же момент.

Работа со значениями

В классах XElement и XAttribute определено свойство Value типа string. Если элемент имеет единственный дочерний узел XText, тогда свойство Value класса XElement действует в качестве удобного сокращения для доступа к содержимому такого узла. В случае класса XAttribute свойство Value — это просто значение атрибута.

Несмотря на отличия в хранении, модель X-DOM предоставляет согласованный набор операций для работы со значениями элементов и атрибутов.

Установка значений

Существуют два способа присваивания значения: вызов метода SetValue или установка свойства Value. Метод SetValue гибче, т.к. он принимает не только строки, но и другие простые типы данных:

```
var e = new XElement ("date", DateTime.Now);
e.SetValue (DateTime.Now.AddDays(1));
Console.Write (e.Value);                                // 2019-10-02T16:39:10.734375+09:00
```

Мы могли бы взамен просто установить свойство Value элемента, но тогда пришлось бы вручную преобразовывать значение DateTime в строку. Такое дей-

ствие сложнее обычного вызова метода `ToString`, потому что требует использования класса `XmlConvert` для получения результата, совместимого с XML.

Когда конструктору класса `XElement` или `XAttribute` передается значение, то же самое автоматическое преобразование выполняется для нестроковых типов. В результате гарантируется корректность форматирования значений `DateTime`; значение `true` записывается в нижнем регистре, а `double.NegativeInfinity` записывается в виде `-INF`.

Получение значений

Чтобы пойти другим путем и разобрать значение `Value` обратно в базовый тип, необходимо лишь привести `XElement` или `XAttribute` к желаемому типу. Прием выглядит так, как будто он не должен работать — но он работает! Например:

```
XElement e = new XElement ("now", DateTime.Now);
DateTime dt = (DateTime) e;

XAttribute a = new XAttribute ("resolution", 1.234);
double res = (double) a;
```

Элемент или атрибут не хранит значения `DateTime` или числа в их естественной форме — они всегда хранятся в виде текста и при необходимости разбираются. Кроме того, исходный тип не запоминается, поэтому приводить нужно корректно, чтобы избежать ошибки во время выполнения. Для повышения надежности кода приведение можно поместить в блок `try/catch`, перехватывающий исключение `FormatException`.

Явные приведения `XElement` и `XAttribute` могут обеспечить разбор в следующие типы:

- все стандартные числовые типы;
- типы `string`, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`;
- версии `Nullable<T>` вышеупомянутых типов значений.

Приведение к типу, допускающему `null`, удобно применять в сочетании с методами `Element` и `Attribute`, т.к. даже если запрошенное имя не существует, то приведение все равно работает. Например, если `x` не имеет элемента `timeout`, тогда первая строка генерирует ошибку во время выполнения, а вторая — нет:

```
int timeout = (int) x.Element ("timeout"); // Ошибка
int? timeout = (int?) x.Element ("timeout"); // Нормально; timeout равно null
```

С помощью операции `??` в финальном результате можно избавиться от типа, допускающего `null`. Если атрибут `resolution` не существует, то переменная `resolution` получит значение `1.0`:

```
double resolution = (double?) x.Attribute ("resolution") ?? 1.0;
```

Тем не менее, приведение к типу, допускающему `null`, не избавит от неприятностей, если элемент или атрибут существует и имеет пустое (либо неправильно сформированное) значение. Для этого придется перехватывать исключение `FormatException`.

Приведения можно также использовать в запросах LINQ. Представленный ниже запрос возвращает John:

```
var data = XElement.Parse (   
    @<data>  
        <customer id='1' name='Mary' credit='100' />  
        <customer id='2' name='John' credit='150' />  
        <customer id='3' name='Anne' />  
    </data>);  
  
IEnumerable<string> query = from cust in data.Elements()  
                               where (int?) cust.Attribute ("credit") > 100  
                               select cust.Attribute ("name").Value;  
  
    Приведение к типу int, допускающему null, позволяет избежать исключения NullReferenceException для заказчика Anne, у которого отсутствует атрибут credit. Другое решение могло бы предусматривать добавление предиката в конструкцию where:  
    where cust.Attributes ("credit").Any() && (int) cust.Attribute...  
    Те же самые принципы применяются при запрашивании значений элементов.
```

Значения и узлы со смешанным содержимым

Имея доступ к значению Value, может возникнуть вопрос о том, нужно ли вообще работать напрямую с узлами XText? Да, если они имеют смешанное содержимое. Например:

```
<summary>An XAttribute is <b>not</b> an XNode</summary>
```

Простого свойства Value для захвата содержимого summary недостаточно. В элементе summary присутствуют три дочерних узла: XText, XElement и XText. Вот как их сконструировать:

```
XElement summary = new XElement ("summary",  
    new XText ("An XAttribute is "),  
    new XElement ("bold", "not"),  
    new XText (" an XNode"));
```

Интересно отметить, что мы по-прежнему можем обращаться к свойству Value элемента summary без генерации исключения. Взамен мы получаем конкатенацию значений всех дочерних узлов:

```
An XAttribute is not an XNode
```

Также разрешено переустанавливать свойство Value элемента summary ценой замены всех предыдущих дочерних узлов единственным новым узлом XText.

Автоматическая конкатенация XText

При добавлении простого содержимого в XElement модель X-DOM дополняет существующий дочерний узел XText, а не создает новый. В следующих примерах e1 и e2 получают только один дочерний элемент XText со значением HelloWorld:

```
var e1 = new XElement ("test", "Hello"); e1.Add ("World");
var e2 = new XElement ("test", "Hello", "World");
```

Однако если узлы XText создаются специально, тогда дочерних элементов будет несколько:

```
var e = new XElement ("test", new XText ("Hello"), new XText ("World"));
Console.WriteLine (e.Value);                                // HelloWorld
Console.WriteLine (e.Nodes().Count());                      // 2
```

Объект XElement не выполняет конкатенацию двух узлов XText, поэтому идентичности объектов узлов предохраняются.

Документы и объявления

XDocument

Как упоминалось ранее, объект XDocument является оболочкой для корневого элемента XElement и позволяет добавлять элемент XDeclaration, инструкции обработки, тип документа и комментарии корневого уровня. Объект XDocument не является обязательным и может быть проигнорирован или опущен: в отличие от DOM-модели W3C он не служит средством объединения всего вместе.

Класс XDocument предлагает те же самые функциональные конструкторы, что и класс XElement. И поскольку он основан на XContainer, в нем также поддерживаются методы AddXXX, RemoveXXX и ReplaceXXX. Тем не менее, в отличие от XElement класс XDocument может принимать только ограниченное содержимое:

- единственный объект XElement (“корень”);
- единственный объект XDeclaration;
- единственный объект XDocumentType (для ссылки на DTD (Document Type Definition — определение типа документа));
- любое количество объектов XProcessingInstruction;
- любое количество объектов XComment.



Для получения допустимого объекта XDocument из всего перечисленного обязательным считается только корневой объект XElement. Объект XDeclaration необязателен — при его отсутствии во время сериализации применяются стандартные настройки.

Простейший допустимый XDocument имеет только корневой элемент:

```
var doc = new XDocument (
    new XElement ("test", "data")
);
```

Обратите внимание, что мы не включили объект XDeclaration. Однако файл, созданный в результате вызова метода doc.Save, будет по-прежнему содержать XML-объявление, потому что оно генерируется по умолчанию.

В следующем примере создается простой, но корректный XHTML-файл, иллюстрирующий все конструкции, которые может принимать XDocument:

```
var styleInstruction = new XProcessingInstruction (
    "xml-stylesheet", "href='styles.css' type='text/css'");
var docType = new XDocumentType ("html",
    "-//W3C//DTD XHTML 1.0 Strict//EN",
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);
XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
    new XElement (ns + "html",
        new XElement (ns + "head",
            new XElement (ns + "title", "An XHTML page")),
        new XElement (ns + "body",
            new XElement (ns + "p", "This is the content")));
);
var doc =
    new XDocument (
        new XDeclaration ("1.0", "utf-8", "no"),
        new XComment ("Reference a stylesheet"),
        styleInstruction,
        docType,
        root);
doc.Save ("test.html");
```

Ниже показано содержимое результирующего файла test.html:

```
<?xml version=1.0 encoding=utf-8 standalone=no?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns=http://www.w3.org/1999/xhtml>
<head>
    <title>An XHTML page</title>
</head>
<body>
    <p>This is the content</p>
</body>
</html>
```

Класс XDocument имеет свойство Root, которое служит сокращением для доступа к единственному объекту XElement документа. Обратная ссылка предоставляется свойством Document класса XObject, которая работает для всех объектов в дереве:

```
Console.WriteLine (doc.Root.Name.LocalName);           // html
 XElement bodyNode = doc.Root.Element (ns + "body");
 Console.WriteLine (bodyNode.Document == doc);          // True
```

Вспомните, что дочерние узлы документа не имеют родительского элемента:

```
Console.WriteLine (doc.Root.Parent == null);           // True
foreach (XNode node in doc.Nodes())
    Console.Write (node.Parent == null);                // TrueTrueTrueTrue
```



Объект `XDeclaration` — это не `XNode`, и в отличие от комментариев, инструкций обработки и корневого элемента он не должен присутствовать внутри коллекции `Nodes` документа. Взамен он присваивается отдельному свойству по имени `Declaration`. Именно потому в последнем примере значение `True` в выводе повторялось четыре раза, а не пять.

Объявления XML

Стандартный XML-файл начинается с примерно такого объявления:

```
<?xml version=1.0 encoding=utf-8 standalone=yes?>
```

Объявление XML гарантирует, что содержимое файла будет корректно разобрано и воспринято средством чтения. При выдаче объявлений XML объекты `XElement` и `XDocument` следуют описанным ниже правилам:

- вызов метода `Save` с именем файла всегда записывает объявление;
- вызов метода `Save` с экземпляром `XmlWriter` записывает объявление, если только `XmlWriter` не был проинструментирован иначе;
- метод `ToString` никогда не выдает объявление XML.



Объект `XmlWriter` можно проинструментировать о том, что он не должен генерировать объявление, путем установки свойств `OmitXmlDeclaration` и `ConformanceLevel` объекта `XmlWriterSettings` при конструировании экземпляра `XmlWriter`. Об этом пойдет речь в главе 11.

Наличие или отсутствие объекта `XDeclaration` никак не влияет на то, записывается объявление XML либо нет. Объект `XDeclaration` предназначен для предоставления подсказок XML-сериализации двояким образом:

- какую кодировку текста использовать;
- что именно помещать в атрибуты `encoding` и `standalone` объявления XML (которые должны записываться объявлением).

Конструктор класса `XDeclaration` принимает три аргумента, которые соответствуют атрибутам `version`, `encoding` и `standalone`. В следующем примере содержимое `test.xml` кодируется с применением UTF-16:

```
var doc = new XDocument (
    new XDeclaration ("1.0", "utf-16", "yes"),
    new XElement ("test", "data")
);
doc.Save ("test.xml");
```



Что бы ни было указано для версии XML, средство записи XML его игнорирует, всегда записывая "1.0".

Кодировка должна указываться с использованием кода IETF, подобного "utf-16" — в точности, как он будет представлен в объявлении XML.

Запись объявления в строку

Предположим, что объект `XDocument` необходимо сериализовать в строку, включая объявление XML. Поскольку метод `ToString` не записывает объявление, мы должны применять вместо него `XmlWriter`:

```
var doc = new XDocument (
    new XDeclaration ("1.0", "utf-8", "yes"),
    new XElement ("test", "data")
);
var output = new StringBuilder();
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create (output, settings))
    doc.Save (xw);
Console.WriteLine (output.ToString());
```

Вот результат:

```
<?xml version=1.0 encoding=utf-16 standalone=yes?>
<test>data</test>
```

Обратите внимание, что в выводе получена кодировка UTF-16 — несмотря на то, что в `XDeclaration` была явно затребована кодировка UTF-8! Результат может выглядеть как ошибка, но на самом деле объект `XmlWriter` удивительно интеллектуален. Из-за того, что запись производится в строку, а не в файл или поток, невозможно использовать никакую другую кодировку кроме UTF-16 — формат, в котором внутренне хранятся строки. Таким образом, `XmlWriter` записывает "utf-16", так что никакого обмана здесь нет.

Это также объясняет причину, по которой метод `ToString` не выпускает объявление XML. Представьте, что вместо вызова метода `Save` для записи `XDocument` в файл вы поступаете следующим образом:

```
File.WriteAllText ("data.xml", doc.ToString());
```

Как уже утверждалось, в файле `data.xml` будет отсутствовать объявление XML, делая его незавершенным, однако по-прежнему поддающимся разбору (кодировка текста может быть выведена). Но если бы метод `ToString` выдавал объявление XML, тогда файл `data.xml` фактически содержал бы некорректное объявление (`encoding="utf-16"`), которое могло бы препятствовать его успешному чтению, потому что метод `WriteAllText` кодирует с применением UTF-8.

Имена и пространства имен

Точно так же как типы .NET могут иметь пространства имен, то же самое возможно для элементов и атрибутов XML.

Пространства имен XML преследуют две цели. Во-первых, подобно пространствам имен в языке C# они помогают избегать конфликтов имен. Такая проблема может возникать при слиянии данных из нескольких XML-файлов.

Во-вторых, пространства имен придают имени абсолютный смысл. Например, имя “nil” может означать все что угодно. Тем не менее, в рамках пространства имен `http://www.w3.org/2001/XMLSchema-instance` имя “nil” означает своего рода эквивалент значения `null` в C# и сопровождается специальными правилами его использования.

Поскольку пространства имен XML являются весомым источником путаницы, мы раскроем данную тему сначала в общих чертах и затем перейдем к применению пространств имен в LINQ to XML.

Пространства имен в XML

Предположим, что требуется определить элемент `customer` в пространстве имен `OReilly.Nutshell.CSharp`. Сделать это можно двумя способами. Первый способ — воспользоваться атрибутом `xmlns`:

```
<customer xmlns=OReilly.Nutshell.CSharp/>
```

`xmlns` представляет собой специальный зарезервированный атрибут. В случае применения в подобной манере он выполняет две функции:

- указывает пространство имен для данного элемента;
- указывает стандартное пространство имен для всех элементов-потомков.

Таким образом, в следующем примере `address` и `postcode` неявно находятся в пространстве имен `OReilly.Nutshell.CSharp`:

```
<customer xmlns=OReilly.Nutshell.CSharp>
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

Если нужно, чтобы `address` и `postcode` не имели пространства имен, тогда понадобится поступить так:

```
<customer xmlns=OReilly.Nutshell.CSharp>
  <address xmlns="">
    <postcode>02138</postcode>  <!-- postcode теперь наследует пустое
пространство имен --&gt;
  &lt;/address&gt;
&lt;/customer&gt;</pre>
```

Префиксы

Другой способ указания пространства имен предусматривает использование префикса. Префикс — это псевдоним, который назначается пространству имен с целью сокращения клавиатурного ввода. С применением префикса связаны два шага — определение префикса и его использование. Шаги можно объединить:

```
<nut:customer xmlns:nut=OReilly.Nutshell.CSharp/>
```

Здесь происходят два разных действия. В правой части конструкция `xmlns:nut="..."` определяет префикс по имени `nut` и делает его доступным данному элементу и всем его потомкам. В левой части конструкция `nut:customer` назначает вновь выделенный префикс элементу `customer`.

Элемент, снабженный префиксом, не определяет стандартное пространство имен для потомков. В следующем XML-коде `firstname` имеет пустое пространство имен:

```
<nut:customer xmlns:nut=OReilly.Nutshell.CSharp>
  <firstname>Joe</firstname>
</customer>
```

Чтобы назначить `firstname` пространство имен `OReilly.Nutshell.CSharp`, потребуется поступить так:

```
<nut:customer xmlns:nut=OReilly.Nutshell.CSharp>
  <nut:firstname>Joe</firstname>
</customer>
```

Префикс — или префиксы — можно также определять для удобства работы с потомками, не назначая любой из этих префиксов самому родительскому элементу. В показанном ниже коде определены два префикса, `i` и `z`, а сам элемент `customer` оставлен с пустым пространством имен:

```
<customer xmlns:i=http://www.w3.org/2001/XMLSchema-instance
  xmlns:z=http://schemas.microsoft.com/2003/10/Serialization/>
  ...
</customer>
```

Если бы узел был корневым, то `i` и `z` были бы доступны всему документу. Префиксы удобны, когда элементы должны извлекаться из нескольких пространств имен.

Обратите внимание, что в рассмотренном примере оба пространства имен представляют собой URI. Применение URI (которыми вы владеете) является стандартной практикой: оно обеспечивает уникальность пространств имен. Таким образом, в реальных обстоятельствах элемент `customer`, скорее всего, будет больше похож на:

```
<customer xmlns=http://oreilly.com/schemas/nutshell/csharp/>
```

или на:

```
<nut:customer xmlns:nut=http://oreilly.com/schemas/nutshell/csharp/>
```

Атрибуты

Назначать пространства имен можно также и атрибутам. Главное отличие состоит в том, что атрибут всегда требует префикса. Например:

```
<customer xmlns:nut=OReilly.Nutshell.CSharp nut:id=123 />
```

Еще одно отличие связано с тем, что неуточненный атрибут всегда имеет пустое пространство имен: он никогда не наследует стандартное пространство имен от своего родительского элемента.

Как правило, атрибуты не нуждаются в пространствах имен, потому что их смысл обычно локален по отношению к элементу. Исключение составляют универсальные атрибуты или атрибуты метаданных, такие как атрибут `nil`, определенный W3C:

```
<customer xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance>
<firstname>Joe</firstname>
<lastname xsi:nil=true/>
</customer>
```

Здесь однозначно указано на то, что `lastname` является `nil` (`null` в C#), а не пустой строкой. Поскольку мы используем стандартное пространство имен, универсальная утилита разбора может точно определить наше намерение.

Указание пространств имен в X-DOM

До сих пор в настоящей главе в качестве имен `XElement` и `XAttribute` применялись только простые строки. Простая строка соответствует имени XML с пустым пространством имен, что очень похоже на тип .NET, определенный в глобальном пространстве имен.

Существует пара подходов к указанию пространства имен XML. Первый из них — заключение его в фигурные скобки и помещение перед локальным именем:

```
var e = new XElement ("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine (e.ToString());
```

Вот результирующий XML-код:

```
<customer xmlns=http://domain.com/xmlspace>Bloggs</customer>
```

Второй (и более эффективный) подход предусматривает использование типов `XNamespace` и `XName`. Их определения показаны ниже:

```
public sealed class XNamespace
{
    public string NamespaceName { get; }
}
public sealed class XName // Локальное имя с дополнительным пространством имен
{
    public string LocalName { get; }
    public XNamespace Namespace { get; } // Необязательно
}
```

Оба типа определяют неявные приведения от `string`, поэтому следующий код вполне законен:

```
XNamespace ns      = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  = "{http://domain.com/xmlspace}customer";
```

В типе `XNamespace` также перегружена операция `+`, что позволяет комбинировать пространство имен с именем в `XName`, не применяя фигурные скобки:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine (fullName); // {http://domain.com/xmlspace}customer
```

Все конструкторы и методы в модели X-DOM, которые принимают имя элемента или атрибута, на самом деле принимают объект `XName`, а не строку. Причина того, что строку можно заменять (как во всех примерах, приведенных до настоящего момента), связана с неявным приведением.

Пространство имен указывается одинаково независимо от того, чем является сущность — элементом или атрибутом:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
    new XAttribute (ns + "id", 123)
);
```

Модель X-DOM и стандартные пространства имен

Модель X-DOM игнорирует концепцию стандартного пространства имен до тех пор, пока не наступает момент фактического вывода XML. Это означает, что когда вы конструируете дочерний элемент XElement, то при необходимости должны предоставить его пространство имен явно: оно не будет наследоваться от родительского элемента:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
    new XElement (ns + "customer", "Bloggs"),
    new XElement (ns + "purchase", "Bicycle")
);
```

Однако при чтении и выводе XML модель X-DOM использует стандартные пространства имен:

```
Console.WriteLine (data.ToString());
ВЫВОД:
<data xmlns="http://domain.com/xmlspace">
  <customer>Bloggs</customer>
  <purchase>Bicycle</purchase>
</data>
```

```
Console.WriteLine (data.Element (ns + "customer").ToString());
ВЫВОД:
```

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Если дочерние узлы XElement конструируются без указания пространств имен — другими словами, так:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
    new XElement ("customer", "Bloggs"),
    new XElement ("purchase", "Bicycle")
);
Console.WriteLine (data.ToString());
```

тогда будет получен отличающийся результат:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns=>Bloggs</customer>
  <purchase xmlns=>Bicycle</purchase>
</data>
```

Еще одна проблема возникает, когда по причине забывчивости не включено пространство имен при навигации по дереву X-DOM:

```

XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement (ns + "data",
    new XElement (ns + "customer", "Bloggs"),
    new XElement (ns + "purchase", "Bicycle")
);
 XElement x = data.Element (ns + "customer");      // Нормально
 XElement y = data.Element ("customer");           // null

```

Если вы строите дерево X-DOM, не указывая пространства имен, то можете впоследствии назначить любому элементу одиночное пространство имен, как показано ниже:

```

foreach ( XElement e in data.DescendantsAndSelf())
    if (e.Name.Namespace == "")
        e.Name = ns + e.Name.LocalName;

```

Префиксы

Модель X-DOM трактует префиксы точно так же, как пространства имен: чисто как функцию сериализации. Следовательно, вы можете полностью игнорировать проблему префиксов — и это сойдет вам с рук! Единственная причина, по которой вы можете решить поступить иначе, касается эффективности при выводе в XML-файл. Например, взгляните на приведенный далее код:

```

XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";

var mix = new XElement (ns1 + "data",
    new XElement (ns2 + "element", "value"),
    new XElement (ns2 + "element", "value"),
    new XElement (ns2 + "element", "value")
);

```

По умолчанию XElement будет сериализовать результат следующим образом:

```

<data xmlns=http://domain.com/space1>
<element xmlns=http://domain.com/space2>value</element>
<element xmlns=http://domain.com/space2>value</element>
<element xmlns=http://domain.com/space2>value</element>
</data>

```

Как видите, присутствует излишнее дублирование. Решение заключается в том, чтобы не изменять способ конструирования X-DOM, а взамен предоставить сериализатору подсказки перед записью XML. Для этого необходимо добавить атрибуты, определяющие префиксы, которые должны быть применены. Обычно такие атрибуты добавляются к корневому элементу:

```

mix.SetAttributeValue (XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue (XNamespace.Xmlns + "ns2", ns2);

```

Приведенный выше код назначает префикс ns1 переменной ns1 из XNamespace и префикс ns2 переменной ns2. Во время сериализации модель X-DOM автоматически выбирает указанные атрибуты и использует их для уплотнения результирующего XML. Вот результат вызова метода ToString на mix:

```
<ns1:data xmlns:ns1=http://domain.com/space1
           xmlns:ns2=http://domain.com/space2>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Предиксы не изменяют способа конструирования, запрашивания или обновления модели X-DOM — для таких действий мы игнорируем наличие префиксов и продолжаем применять полные имена. Префиксы вступают в игру только при преобразовании в файлы или потоки XML и из них.

Предиксы также учитываются в атрибутах сериализации. В приведенном ниже примере мы записываем дату рождения и кредит заказчика в виде "nil", используя стандартный атрибут W3C. Выделенная строка гарантирует, что префикс сериализуется без нежелательного повторения пространства имен:

```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute (xsi + "nil", true);

var cust = new XElement ("customers",
    new XAttribute (XNamespace.Xmlns + "xsi", xsi),
    new XElement ("customer",
        new XElement ("lastname", "Bloggs"),
        new XElement ("dob", nil),
        new XElement ("credit", nil)
    )
);
```

А вот результирующий XML-код:

```
<customers xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance>
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil=true />
    <credit xsi:nil=true />
  </customer>
</customers>
```

Для краткости мы предварительно объявили пустой XAttribute, так что его можно применять два раза при построении DOM-модели. Дважды ссылаться на тот же самый атрибут разрешено из-за того, что при необходимости он автоматически дублируется.

Аннотации

С помощью аннотаций к любому объекту XObject можно присоединять специальные данные. Аннотации предназначены для вашего личного использования и трактуются моделью X-DOM как “черные ящики”. Если вы когда-либо имели дело со свойством Tag какого-нибудь элемента управления Windows Forms или Windows Presentation Foundation (WPF), то концепция должна быть знакомой — лишь с тем отличием, что разрешено иметь множество аннотаций и назначать им закрытую область видимости. Допускается создавать аннота-

цию, которую другие типы не смогут даже видеть, не говоря уже о том, чтобы перезаписывать.

За добавление и удаление аннотаций отвечают следующие методы класса XObject:

```
public void AddAnnotation (object annotation)
public void RemoveAnnotations<T> ()      where T : class
```

Перечисленные далее методы извлекают аннотации:

```
public T Annotation<T> ()                  where T : class
public IEnumerable<T> Annotations<T> () where T : class
```

Каждой аннотации назначается ключ согласно ее типу, который должен быть ссылочным. Показанный ниже код добавляет и затем извлекает аннотацию типа string:

```
XElement e = new XElement ("test");
e.AddAnnotation ("Hello");
Console.WriteLine (e.Annotation<string>()); // Hello
```

Можно добавить множество аннотаций того же самого типа, а затем применить метод Annotations для извлечения последовательности совпадений.

Тем не менее, открытый тип вроде string не обеспечивает создание эффективного ключа, т.к. код в других типах может стать помехой вашим аннотациям. Более удачный подход предполагает использование внутреннего или (вложенного) закрытого класса:

```
class X
{
    class CustomData { internal string Message; } // Закрытый вложенный тип
    static void Test()
    {
        XElement e = new XElement ("test");
        e.AddAnnotation (new CustomData { Message = "Hello" } );
        Console.Write (e.Annotations<CustomData>().First().Message); // Hello
    }
}
```

Для удаления аннотаций вы должны иметь доступ также и к типу ключа:

```
e.RemoveAnnotations<CustomData>();
```

Проектирование в модель X-DOM

До сих пор мы показывали, как применять LINQ для получения данных из модели X-DOM. Запросы LINQ можно также использовать для проектирования в модель X-DOM. Источником может быть все, к чему поддерживаются запросы LINQ, в том числе:

- сущностные классы EF Core;
- локальная коллекция;
- другая модель X-DOM.

Независимо от источника применяется та же самая стратегия, что и в случае использования LINQ для выдачи дерева X-DOM: сначала записывается выражение функционального построения, которое создает желаемую форму X-DOM, а затем на основе этого выражения строится запрос LINQ. Например, пусть необходимо извлекать заказчиков из базы данных в XML-код следующего вида:

```
<customers>
  <customer id=1>
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

Мы начинаем с того, что представляем выражение функционального построения для X-DOM, применяя простые литералы:

```
var customers =
  new XElement ("customers",
    new XElement ("customer", new XAttribute ("id", 1),
      new XElement ("name", "Sue"),
      new XElement ("buys", 3)
    )
  );
```

Затем мы превращаем это выражение в проекцию и строим на его основе запрос LINQ:

```
var customers =
  new XElement ("customers",
    // Из-за дефекта в EF Core мы должны вызывать метод AsEnumerable
    from c in dbContext.Customers.AsEnumerable()
    select
      new XElement ("customer", new XAttribute ("id", c.ID),
        new XElement ("name", c.Name),
        new XElement ("buys", c.Purchases.Count)
      )
  );
```



Вызов метода AsEnumerable требуется из-за дефекта в EF Core (исправление запланировано в более позднем выпуске). После того, как дефект будет устранен, удаление вызова AsEnumerable увеличит эффективность, предотвращая обращение к базе данных при каждом вызове c.Purchases.Count.

Ниже показан результат:

```
<customers>
  <customer id=1>
    <name>Tom</name>
    <buys>3</buys>
  </customer>
  <customer id=2>
    <name>Harry</name>
    <buys>2</buys>
  </customer>
  ...
</customers>
```

Чтобы лучше понять, как все работает, сконструируем тот же самый запрос за два шага. Вот первый шаг:

```
 IEnumerable< XElement > sqlQuery =  
     from c in dbContext.Customers.AsEnumerable()  
     select  
         new XElement ("customer", new XAttribute ("id", c.ID),  
             new XElement ("name", c.Name),  
             new XElement ("buys", c.Purchases.Count)  
 );
```

Внутренняя порция представляет собой нормальный запрос LINQ, который выполняет проецирование в элементы XElement. Вот второй шаг:

```
 var customers = new XElement ("customers", sqlQuery);
```

Здесь конструируется корневой элемент XElement. Единственный необычный аспект заключается в том, что содержимое, т.е. sqlQuery — это не одиничный XElement, а реализация IQueryables< XElement >, которая в свою очередь реализует интерфейс IEnumerable< XElement >. Вспомните, что при обработке XML-содержимого происходит автоматическое перечисление коллекций. Таким образом, каждый элемент XElement добавляется как дочерний узел.

Устранение пустых элементов

Предположим, что в предыдущем примере также необходимо включить подробности о последней дорогой покупке заказчика. Решить задачу можно было бы следующим образом:

```
 var customers =  
     new XElement ("customers",  
         // После устранения дефекта в EF Core вызов AsEnumerable можно удалить  
         from c in dbContext.Customers.AsEnumerable()  
         let lastBigBuy = (from p in c.Purchases  
                           where p.Price > 1000  
                           orderby p.Date descending  
                           select p).FirstOrDefault()  
         select  
             new XElement ("customer", new XAttribute ("id", c.ID),  
                 new XElement ("name", c.Name),  
                 new XElement ("buys", c.Purchases.Count),  
                 new XElement ("lastBigBuy",  
                     new XElement ("description", lastBigBuy?.Description,  
                     new XElement ("price", lastBigBuy?.Price ?? 0m)  
                 )  
             )  
     );
```

Однако здесь будут выпускаться пустые элементы для заказчиков, не совершивших дорогих покупок. (Если бы это был локальный запрос, а не запрос к базе данных, то сгенерировалось бы исключение NullReferenceException.) В таких случаях лучше полностью опустить узел lastBigBuy, поместив конструктор для элемента lastBigBuy внутрь условной операции: