

Тем не менее, при перечислении результатов выполнение продолжается несколько иначе, чем в случае обычного последовательного запроса. Последовательный запрос поддерживается полностью потребителем с применением модели с пассивным источником: каждый элемент извлекается из входной последовательности только тогда, когда он затребован потребителем.

Параллельный запрос обычно использует независимые потоки для извлечения элементов из входной последовательности, причем с небольшим *упреждением*, до того момента, когда они понадобятся потребителю (почти как телесуфлер у дикторов новостей или буфер в проигрывателях компакт-дисков). Затем он обрабатывает элементы параллельно через цепочку запросов, удерживая результаты в небольшом буфере, чтобы они были готовы при затребовании потребителем. Если потребитель приостанавливает или прекращает перечисление до его завершения, обработчик запроса тоже приостанавливается или прекращает работу, чтобы не тратить впустую время ЦП или память.



Поведение буферизации PLINQ можно настраивать, вызывая метод `WithMergeOptions` после `AsParallel`. Стандартное значение `AutoBuffered` перечисления `ParallelMergeOptions` обычно дает наилучшие окончательные результаты. Значение `NotBuffered` отключает буфер и полезно в ситуации, когда результаты необходимо увидеть как можно скорее; значение `FullyBuffered` кеширует целый результирующий набор перед представлением его потребителю (подобным образом изначально работают операции `OrderBy` и `Reverse`, а также операции над элементами, операции агрегирования и операции преобразования).

PLINQ и упорядочивание

Побочный эффект от распараллеливания операций запросов заключается в том, что когда результаты объединены, они не обязательно располагаются в том же самом порядке, в котором они были получены (см. рис. 22.2). Другими словами, обычная гарантия предохранения порядка LINQ для последовательностей больше не поддерживается.

Если нужно предохранить порядок, тогда после вызова `AsParallel` понадобится вызвать метод `AsOrdered`:

```
myCollection.AsParallel().AsOrdered()...
```

Вызов метода `AsOrdered` оказывает влияние на производительность, поскольку инфраструктура PLINQ должна отслеживать исходные позиции всех элементов.

Позже последствия от вызова `AsOrdered` в запросе можно отменить, вызвав метод `AsUnordered`: это вводит “точку случайного тасования”, которая после ее прохождения позволяет запросу выполняться более эффективно. Таким образом, если необходимо предохранить упорядочение входной последовательности только для первых двух операций запросов, то можно поступить так:

```
inputSequence.AsParallel().AsOrdered()
    .QueryOperator1()
    .QueryOperator2()
    .AsUnordered() // Начиная с этой точки, упорядочивание роли не играет
    .QueryOperator3()
    ...
```

Метод `AsOrdered` не является стандартным вариантом, потому что для большинства запросов первоначальное упорядочивание во входной последовательности не имеет значения. Другими словами, если бы метод `AsOrdered` использовался по умолчанию, то к большинству параллельных запросов пришлось бы применять метод `AsUnordered`, чтобы добиться лучших показателей производительности, и поступать так было бы обременительно.

Ограничения PLINQ

Существует несколько практических ограничений относительно того, что инфраструктура PLINQ способна распараллеливать. Следующие операции запросов по умолчанию предотвращают распараллеливание, если только исходные элементы не находятся в своих первоначальных индексных позициях:

- индексированные версии `Select`, `SelectMany` и `ElementAt`.

Большинство операций запросов изменяют индексные позиции элементов (включая операции, удаляющие элементы, такие как `Where`). Это означает, что если нужно использовать предшествующие операции, то они обычно должны находиться в начале запроса.

Перечисленные ниже операции запросов допускают распараллеливание, но применяют затратную стратегию разбиения, которая иногда может оказываться медленнее последовательной обработки:

- `Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect` и `Except`.

Перегруженные версии операции `Aggregate`, принимающие начальное значение (в аргументе `seed`), в своем стандартном виде не поддерживают возможность распараллеливания — в PLINQ для такой цели предлагаются специальные перегруженные версии (см. раздел “Оптимизация PLINQ” далее в главе).

Все остальные операции поддаются распараллеливанию, хотя их использование не гарантирует, что запрос будет распараллелен. Инфраструктура PLINQ может выполнять запрос последовательно, если ожидает, что накладные расходы от распараллеливания приведут к замедлению имеющегося конкретного запроса. Такое поведение можно переопределить и принудительно применять параллелизм, вызвав показанный ниже метод после `AsParallel`:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Пример: параллельная программа проверки орфографии

Предположим, что требуется написать программу проверки орфографии, которая выполняется быстро для очень больших документов за счет использования всех свободных процессорных ядер. Выразив алгоритм в виде запроса LINQ, мы можем его легко распараллелить.

Первый шаг предусматривает загрузку словаря английских слов в объект `HashSet`, чтобы обеспечить эффективный поиск:

```
if (!File.Exists ("WordLookup.txt"))           // Содержит около 150 000 слов
    File.WriteAllText ("WordLookup.txt",
        await new HttpClient().GetStringAsync (
            "http://www.albahari.com/ispell/allwords.txt"));
var wordLookup = new HashSet<string> (
    File.ReadAllLines ("WordLookup.txt"),
    StringComparer.InvariantCultureIgnoreCase);
```

Затем мы будем применять полученное средство поиска слов для создания тестового “документа”, содержащего массив из миллиона случайных слов. После построения массива мы внесем пару орфографических ошибок:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();
string[] wordsToTest = Enumerable.Range (0, 1000000)
    .Select (i => wordList [random.Next (0, wordList.Length)])
    .ToArray();
wordsToTest [12345] = "woozsh";           // Внесение пары
wordsToTest [23456] = "wubsie";          // орфографических ошибок.
```

Теперь мы можем выполнить параллельную проверку орфографии, сверяя `wordsToTest` с `wordLookup`. PLINQ позволяет делать это очень просто:

```
var query = wordsToTest
    .AsParallel ()
    .Select ((word, index) => (word, index))
    .Where (iword => !wordLookup.Contains (iword.word))
    .OrderBy (iword => iword.index);
foreach (var mistake in query)
    Console.WriteLine (mistake.word + " - index = " + mistake.index);
```

Вот вывод:

```
woozsh - index = 12345
wubsie - index = 23456
```



Обратите внимание, что в запросе используются кортежи (`word, index`), а не анонимные типы. Поскольку кортежи реализованы как типы значений, а не ссылочные типы, это снижает пиковое потребление памяти и повышает производительность за счет сокращения выделений памяти в куче и последующей сборки мусора. (Сравнительный анализ показывает, что на практике выигрыш будет умеренным из-за эффективности диспетчера памяти и того факта, что рассматриваемые выделения памяти не сохраняются за рамками поколения 0.)

Использование `ThreadLocal<T>`

Давайте расширим наш пример, распараллелив само создание случайного тестового списка слов. Мы структурировали его как запрос LINQ, так что все должно быть легко. Вот последовательная версия:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
    .Select (i => wordList [random.Next (0, wordList.Length)])
    .ToArray();
```

К сожалению, вызов метода `random.Next` небезопасен в отношении потоков, поэтому работа не сводится к простому добавлению в запрос вызова `AsParallel`. Потенциальным решением может быть написание функции, помещающей вызов `random.Next` внутрь блокировки, но это ограничило бы параллелизм. Более удачный вариант предусматривает применение класса `ThreadLocal<Random>` (см. раздел “Локальное хранилище потока” в главе 21) с целью создания отдельного объекта `Random` для каждого потока. Тогда распараллелить запрос можно следующим образом:

```
var localRandom = new ThreadLocal<Random>
    ( () => new Random (Guid.NewGuid ().GetHashCode ()) );
string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
    .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
    .ToArray();
```

В нашей фабричной функции для создания объекта `Random` мы передаем хеш-код `Guid`, гарантируя тем самым, что даже если два объекта `Random` создаются в рамках короткого промежутка времени, то они все равно будут выдавать отличающиеся последовательности случайных чисел.

Когда необходимо использовать PLINQ?

Довольно заманчиво поискать в существующих приложениях запросы LINQ и поэкспериментировать с их распараллеливанием. Однако обычно это не-продуктивно, т.к. большинство задач, для которых LINQ является очевидным наилучшим решением, выполняются очень быстро, а потому не выигрывают от распараллеливания. Более удачный подход предполагает поиск узких мест, интенсивно использующих ЦП, и выяснение, могут ли они быть выражены в виде запроса LINQ. (Приятный побочный эффект от такой ре-структуризации состоит в том, что LINQ обычно делает код более кратким и читабельным.)

Инфраструктура PLINQ хорошо подходит для естественно параллельных задач. Однако она может быть плохим выбором для обработки изображений, потому что объединение миллионов пикселей в выходную последовательность создаст узкое место. Взамен пиксели лучше записывать прямо в массив или блок неуправляемой памяти и применять класс `Parallel` либо параллелизм задач для управления многопоточностью. (Тем не менее, объединение результатов можно аннулировать с использованием `ForAll` — мы обсудим данную тему в разделе “Оптимизация PLINQ” далее в главе. Поступать так имеет смысл, если алгоритм обработки изображений естественным образом приспособливается к LINQ.)

Функциональная чистота

Поскольку PLINQ запускает ваш запрос в параллельных потоках, вы должны избегать выполнения небезопасных к потокам операций. В частности, запись в переменные порождает *побочные эффекты* и потому не является безопасной в отношении потоков:

```
// Следующий запрос умножает каждый элемент на его позицию.  
// Получив на входе Enumerable.Range(0, 999), он должен  
// вывести последовательность квадратов.  
int i = 0;  
var query = from n in Enumerable.Range(0, 999).AsParallel() select n * i++;
```

Мы могли бы сделать инкрементирование переменной *i* безопасным к потокам за счет применения блокировок, но все еще останется проблема того, что *i* не обязательно будет соответствовать позиции входного элемента. И добавление *AsOrdered* в запрос не решит последнюю проблему, т.к. метод *AsOrdered* гарантирует лишь то, что элементы выводятся в порядке, согласованном с порядком, который они имели бы при последовательной обработке — он не осуществляет действительную их обработку последовательным образом.

Взамен данный запрос должен быть переписан с использованием индексированной версии *Select*:

```
var query = Enumerable.Range(0, 999).AsParallel().Select ((n, i) => n * i);
```

Для достижения лучшей производительности любые методы, вызываемые из операций запросов, должны быть безопасными к потокам, не производя запись в поля или свойства (не давать побочные эффекты, т.е. быть функционально чистыми). Если они являются безопасными в отношении потоков благодаря блокированию, тогда потенциал параллелизма запроса будет ограничен последствиями соперничества.

Установка степени параллелизма

По умолчанию PLINQ выбирает оптимальную степень параллелизма для задействованного процессора. Ее можно переопределить, вызвав метод *WithDegreeOfParallelism* после *AsParallel*:

```
...AsParallel().WithDegreeOfParallelism(4)...
```

Примером, когда степень параллелизма может быть увеличена до значения, превышающего количество ядер, является работа с интенсивным вводом-выводом (скажем, загрузка множества веб-страниц за раз). Тем не менее, комбинаторы задач и асинхронные функции предлагают аналогично несложное, но более эффективное решение (см. раздел “Комбинаторы задач” в главе 14). В отличие от объектов *Task* инфраструктура PLINQ не способна выполнять работу с интенсивным вводом-выводом без блокирования потоков (и что еще хуже — потоков из пула).

Изменение степени параллелизма

Метод `WithDegreeOfParallelism` можно вызывать только один раз внутри запроса PLINQ. Если его необходимо вызвать снова, то потребуется принудительно инициировать слияние и повторное разбиение запроса, еще раз вызвав метод `AsParallel` внутри запроса:

```
"The Quick Brown Fox"
    .AsParallel().WithDegreeOfParallelism (2)
    .Where (c => !char.IsWhiteSpace (c))
    .AsParallel().WithDegreeOfParallelism (3) // Инициировать слияние
                                                // и разбиение
    .Select (c => char.ToUpper (c))
```

Отмена

Отменить запрос PLINQ, результаты которого потребляются в цикле `foreach`, легко: нужно просто прекратить цикл `foreach` и запрос будет автоматически отменен по причине неявного освобождения перечислителя.

Отменить запрос, который заканчивается операцией преобразования, операцией над элементами или операцией агрегирования, можно из другого потока через признак отмены (см. раздел “Отмена” в главе 14). Чтобы вставить такой признак, необходимо после вызова `AsParallel` вызвать метод `WithCancellation`, передав ему свойство `Token` объекта `CancellationTokenSource`. Затем другой поток может вызвать метод `Cancel` на источнике признака (или мы вызовем его самостоятельно с задержкой), что приведет к генерации исключения `OperationCanceledException` в потребителе запроса:

```
IEnumerable<int> tenMillion = Enumerable.Range (3, 10_000_000);

var cancelSource = new CancellationTokenSource();
cancelSource.CancelAfter (100);           // Отменить запрос по прошествии
                                         // 100 мс

var primeNumberQuery =
    from n in tenMillion.AsParallel().WithCancellation (cancelSource.Token)
    where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
    select n;

try
{
    // Начать выполнение запроса:
    int[] primes = primeNumberQuery.ToArray();
    // Мы никогда не попадем сюда, потому что другой поток инициирует отмену.
}
catch (OperationCanceledException)
{
    Console.WriteLine ("Query canceled");    // Запрос отменен
}
```

При инициировании отмены PLINQ ожидает завершения каждого рабочего потока со своим текущим элементом перед тем, как закончить запрос. Это означает, что любые внешние методы, которые вызывает запрос, будут выполняться до полного завершения.

Оптимизация PLINQ

Оптимизация на выходной стороне

Одно из преимуществ инфраструктуры PLINQ связано с тем, что она удобно объединяет результаты распараллеленной работы в единую выходную последовательность. Однако иногда все, что в итоге делается с такой последовательностью — выполнение некоторой функции над каждым элементом:

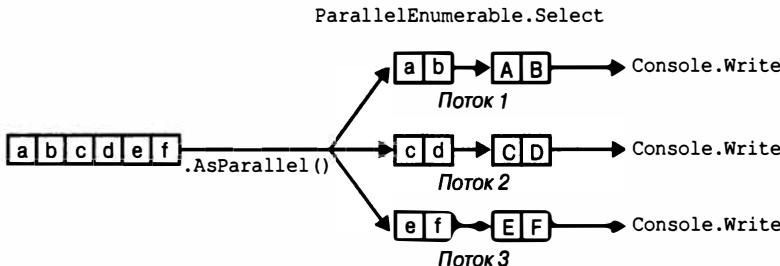
```
foreach (int n in parallelQuery)
    DoSomething (n);
```

В таком случае, если порядок обработки элементов не волнует, тогда эффективность можно улучшить с помощью метода `ForAll` из PLINQ.

Метод `ForAll` запускает делегат для каждого выходного элемента `ParallelQuery`. Он проникает прямо внутрь PLINQ, обходя шаги объединения и перечисления результатов. Ниже приведен простейший пример:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.WriteLine);
```

Процесс продемонстрирован на рис. 22.3.



```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.WriteLine)
```

Рис. 22.3. Метод `ForAll` из PLINQ



Объединение и перечисление результатов — не массовая затратная операция, поэтому оптимизация с помощью `ForAll` дает наибольшую выгоду при наличии большого количества быстро обрабатываемых входных элементов.

Оптимизация на входной стороне

Для назначения входных элементов потокам в PLINQ поддерживаются три стратегии разбиения.

Стратегия	Распределение элементов	Относительная производительность
Разбиение на основе порций	Динамическое	Средняя
Разбиение на основе диапазонов	Статическое	От низкой до очень высокой
Разбиение на основе хеш-кодов	Статическое	Низкая

Для операций запросов, которые требуют сравнения элементов (GroupBy, Join, GroupJoin, Intersect, Except, Union и Distinct), выбор отсутствует: PLINQ всегда использует *разбиение на основе хеш-кодов*. Разбиение на основе хеш-кодов относительно неэффективно в том, что оно требует предварительного вычисления хеш-кода каждого элемента (а потому элементы с одинаковыми хеш-кодами могут обрабатываться в одном и том же потоке). Если вы считаете это слишком медленным, то единственным доступным вариантом будет вызов метода AsSequential с целью отключения распараллеливания.

Для всех остальных операций запросов имеется выбор между разбиением на основе диапазонов и разбиением на основе порций. По умолчанию:

- если входная последовательность *индексируется* (т.е. является массивом или реализует интерфейс `IList<T>`), тогда PLINQ выбирает *разбиение на основе диапазонов*;
- в противном случае PLINQ выбирает *разбиение на основе порций*.

По своей сути разбиение на основе диапазонов выполняется быстрее с длинными последовательностями, для которых каждый элемент требует сходного объема времени ЦП на обработку. В противном случае разбиение на основе порций обычно быстрее.

Чтобы принудительно применить *разбиение на основе диапазонов*, выполните такие действия:

- если запрос начинается с вызова метода `Enumerable.Range`, то замените его вызовом `ParallelEnumerable.Range`;
- иначе просто вызовите метод `ToList` или `ToArray` на входной последовательности (это вполне очевидно повлияет на производительность, что также должно приниматься во внимание).



Метод `ParallelEnumerable.Range` — не просто сокращение для вызова `Enumerable.Range(...).AsParallel()`. Он изменяет производительность запроса, активизируя разбиение на основе диапазонов.

Чтобы принудительно применить разбиение на основе порций, необходимо поместить входную последовательность в вызов `Partitioner.Create` (из пространства имён `System.Collection.Concurrent`) следующим образом:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
    Partitioner.Create(numbers, true).AsParallel()
    .Where(...)
```

Второй аргумент `Partitioner.Create` указывает на то, что для запроса требуется *балансировка нагрузки*, которая представляет собой еще один способ сообщения о выборе разбиения на основе порций.

Разбиение на основе порций работает путем предоставления каждому рабочему потоку возможности периодически захватывать из входной последовательности небольшие “порции” элементов с целью их обработки (рис. 22.4).

Инфраструктура PLINQ начинает с выделения очень маленьких порций (один или два элемента за раз) и затем по мере продвижения запроса увеличивает размер порции: это гарантирует, что небольшие последовательности будут эффективно распараллеливаться, а крупные последовательности не приведут к чрезмерным циклам полного обмена. Если рабочий поток получает “простые” элементы (которые обрабатываются быстро), то в конечном итоге он сможет получить больше порций. Такая система сохраняет каждый поток одинаково занятым (а процессорные ядра “сбалансированными”); единственный недостаток состоит в том, что извлечение элементов из совместно используемой входной последовательности требует синхронизации (обычно монопольной блокировки) — и в результате могут появиться некоторые накладные расходы и состязания.

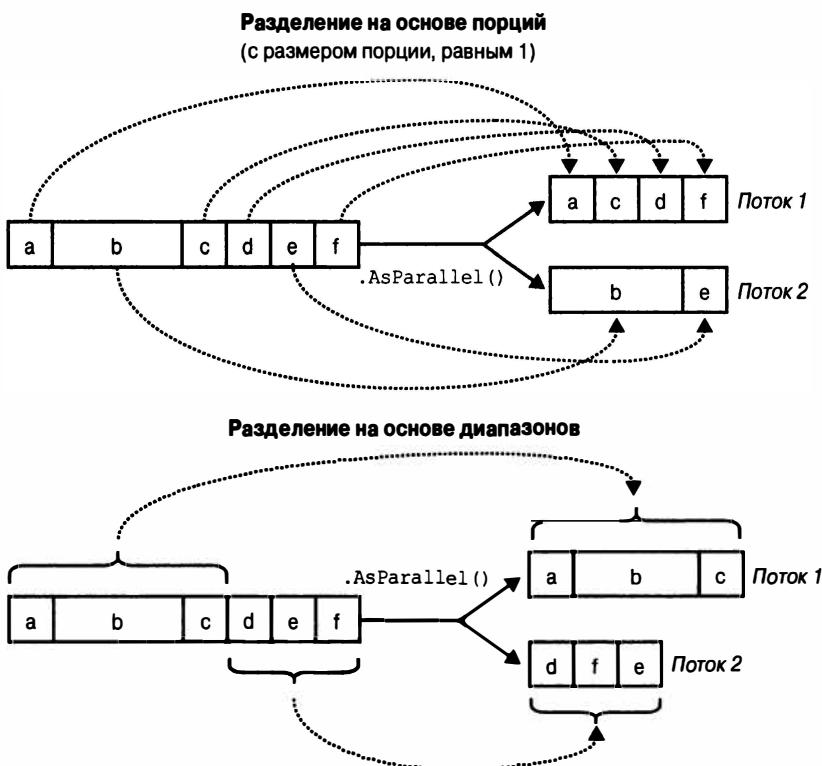


Рис. 22.4. Сравнение разбиения на основе порций и разбиения на основе диапазонов

Разбиение на основе диапазонов пропускает обычное перечисление на входной стороне и предварительно распределяет одинаковое количество элементов для каждого рабочего потока, избегая состязаний на входной последовательности. Но если случится так, что некоторые потоки получат простые элементы и завершатся раньше, то они окажутся в состоянии простого, пока остальные потоки продолжат свою работу. Ранее приведенный пример с простыми числа-

ми может плохо выполняться при разбиении на основе диапазонов. Примером, когда такое разбиение оказывается удачным, является вычисление суммы квадратных корней первых 10 млн целых чисел:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

Метод `ParallelEnumerable.Range` возвращает `ParallelQuery<T>`, поэтому вызывать `AsParallel` впоследствии не придется.



Разбиение на основе диапазонов не обязательно распределяет диапазоны элементов в смежных блоках — взамен может быть выбрана “полосовая” стратегия. Например, при наличии двух рабочих потоков один из них может обрабатывать элементы в нечетных позициях, а другой — элементы в четных позициях. Операция `TakeWhile` почти наверняка инициирует полосовую стратегию, чтобы избежать излишней обработки элементов позже в последовательности.

Оптимизация специального агрегирования

Инфраструктура PLINQ эффективно распараллеливает операции `Sum`, `Average`, `Min` и `Max` без дополнительного вмешательства. Тем не менее, операция `Aggregate` представляет особую трудность для PLINQ. Как было описано в главе 9, операция `Aggregate` выполняет специальное агрегирование. Например, следующий код суммирует последовательность чисел, имитируя операцию `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n); // 6
```

В главе 9 также было показано, что для агрегаций без начальных значений предоставляемый делегат должен быть ассоциативным и коммутативным. Если указанное правило нарушается, тогда инфраструктура PLINQ даст некорректные результаты, поскольку она извлекает множество начальных значений из входной последовательности, чтобы выполнять агрегирование нескольких частей последовательности одновременно.

Агрегации с явными начальными значениями могут выглядеть как безопасный вариант для PLINQ, но, к сожалению, обычно они выполняются последовательно, т.к. полагаются на единственное начальное значение. Чтобы смягчить данную проблему, PLINQ предлагает еще одну перегруженную версию метода `Aggregate`, которая позволяет указывать множество начальных значений — или скорее фабричную функцию начальных значений. В каждом потоке эта функция выполняется для генерации отдельного начального значения, которое фактически становится локальным для потока накопителем, куда локально агрегируются элементы.

Потребуется также предоставить функцию для указания способа объединения локального и главного накопителей. Наконец, перегруженная версия метода `Aggregate` (отчасти беспринципно) ожидает делегат для проведения любой финальной трансформации результата (в принципе его легко обеспечить, просто выполняя нужную функцию на результате после его получения). Таким образом, ниже перечислены четыре делегата в порядке их передачи.

- `seedFactory`. Возвращает новый локальный накопитель.
- `updateAccumulatorFunc`. Агрегирует элемент в локальный накопитель.
- `combineAccumulatorFunc`. Объединяет локальный накопитель с главным накопителем.
- `resultSelector`. Применяет любую финальную трансформацию к конечному результату.



В простых сценариях можно указывать *начальное значение*, а не фабрику начальных значений. Такая тактика потерпит неудачу, когда начальное значение относится к ссылочному типу, который требуется изменять, потому что один и тот же экземпляр будет затем совместно использоваться всеми потоками.

В качестве очень простого примера ниже приведен запрос, который суммирует значения в массиве `numbers`:

```
numbers.AsParallel().Aggregate (
    () => 0,                                     // seedFactory
    (localTotal, n) => localTotal + n,           // updateAccumulatorFunc
    (mainTot, localTot) => mainTot + localTot,   // combineAccumulatorFunc
    finalResult => finalResult)                  // resultSelector
```

Пример несколько надуман, т.к. тот же самый результат не менее эффективно можно было бы получить с применением более простых подходов (скажем, с помощью агрегации без начального значения или, что еще лучше, посредством операции `Sum`). Чтобы предложить более реалистичный пример, предположим, что требуется вычислить частоту появления каждой буквы английского алфавита в заданной строке. Простое последовательное решение может выглядеть следующим образом:

```
string text = "Let's suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
    int index = char.ToUpper (c) - 'A';
    if (index >= 0 && index < 26) letterFrequencies [index]++;
}
```



Примером, когда входной текст может оказаться очень длинным, являются генные цепочки. В таком случае “алфавит” состоит из букв *a, c, g и t*.

Для распараллеливания такого запроса мы могли бы заменить оператор `foreach` вызовом метода `Parallel.ForEach` (как будет показано в следующем разделе), но тогда пришлось бы иметь дело с проблемами параллелизма на совместно используемом массиве. Блокирование доступа к данному массиву решило бы проблемы, но ликвидировало бы возможность распараллеливания.

Операция Aggregate обеспечивает более аккуратное решение. В этом случае накопителем выступает массив, похожий на массив letterFrequencies из предыдущего примера. Ниже представлена последовательная версия, использующая Aggregate:

```
int[] result =
text.Aggregate (
    new int[26], // Создать "накопитель"
    (letterFrequencies, c) => // Агрегировать букву в этот "накопитель"
{
    int index = char.ToUpper (c) - 'A';
    if (index >= 0 && index < 26) letterFrequencies [index]++;
    return letterFrequencies;
});
```

А вот параллельная версия, в которой применяется специальная перегруженная версия Aggregate из PLINQ:

```
int[] result =
text.AsParallel().Aggregate (
() => new int[26], // Создать новый локальный накопитель
(localFrequencies, c) => // Агрегировать в этот локальный накопитель
{
    int index = char.ToUpper (c) - 'A';
    if (index >= 0 && index < 26) localFrequencies [index]++;
    return localFrequencies;
}, // Агрегировать локальный и главный накопители
(mainFreq, localFreq) =>
mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),
finalResult => finalResult // Выполнить любую финальную трансформацию
); // конечного результата
```

Обратите внимание, что функция локального накопителя *изменяет* массив localFrequencies. Возможность выполнения такой оптимизации важна — и она законна, поскольку массив localFrequencies является локальным для каждого потока.

Класс Parallel

Инфраструктура PFX предоставляет базовую форму структурированного параллелизма через три статических метода в классе Parallel.

- Parallel.Invoke. Запускает массив делегатов параллельно.
- Parallel.For. Выполняет параллельный эквивалент цикла for языка C#.
- Parallel.ForEach. Выполняет параллельный эквивалент цикла foreach языка C#.

Все три метода блокируются вплоть до завершения всей работы. Как и с PLINQ, в случае необработанного исключения оставшиеся рабочие потоки останавливаются после их текущей итерации, а исключение (либо их набор) передается обратно вызывающему потоку внутри оболочки AggregateException (как объясняется в разделе “Работа с AggregateException” далее в главе).

Parallel.Invoke

Метод Parallel.Invoke запускает массив делегатов Action параллельно, после чего ожидает их завершения. Его простейшая версия определена следующим образом:

```
public static void Invoke (params Action[] actions);
```

Как и в PLINQ, методы Parallel.* оптимизированы для выполнения работы с интенсивными вычислениями, но не интенсивным вводом-выводом. Тем не менее, загрузка двух веб-страниц за раз позволяет легко продемонстрировать использование метода Parallel.Invoke:

```
Parallel.Invoke (
    () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
    () => new WebClient().DownloadFile ("http://microsoft.com", "ms.html"));
```

На первый взгляд код выглядит удобным сокращением для создания и ожидания двух привязанных к потокам объектов Task. Однако существует важное отличие: метод Parallel.Invoke по-прежнему будет работать эффективно, даже если ему передать массив из миллиона делегатов. Причина в том, что он *разбивает* большое количество элементов на пакеты, которые назначает небольшому числу существующих объектов Task, а не создает отдельный объект Task для каждого делегата.

Как и со всеми методами класса Parallel, объединение результатов возлагается полностью на вас. Это значит, что вы должны помнить о безопасности в отношении потоков. Например, приведенный ниже код не является безопасным к потокам:

```
var data = new List<string>();
Parallel.Invoke (
    () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
    () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Помещение кода добавления в список внутрь блокировки решило бы проблему, но блокировка создаст узкое место в случае гораздо более крупных массивов быстро выполняющихся делегатов. Лучшее решение предусматривает использование безопасных к потокам коллекций, которые рассматриваются далее в главе — идеальным вариантом в данном случае была бы коллекция ConcurrentBag.

Метод Parallel.Invoke также имеет перегруженную версию, принимающую объект ParallelOptions:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

С помощью объекта ParallelOptions можно вставить признак отмены, ограничить максимальную степень параллелизма и указать специальный планировщик задач. Признак отмены играет важную роль, когда выполняется (ориентировочно) большее количество задач, чем имеется процессорных ядер: при отмене все незапущенные делегаты будут отброшены. Однако любые уже выполняющиеся делегаты продолжат свою работу вплоть до ее завершения. В разделе “Отмена” ранее в главе приводился пример применения признаков отмены.

Parallel.For и Parallel.ForEach

Методы Parallel.For и Parallel.ForEach реализуют эквиваленты циклов for и foreach из C#, но с выполнением каждой итерации параллельно, а не последовательно. Ниже показаны их (простейшие) сигнатуры:

```
public static ParallelLoopResult For (
    int fromInclusive, int toExclusive, Action<int> body)
public static ParallelLoopResult ForEach<TSource> (
    IEnumerable<TSource> source, Action<TSource> body)
```

Следующий последовательный цикл for:

```
for (int i = 0; i < 100; i++)
    Foo (i);
```

распараллеливается примерно так:

```
Parallel.For (0, 100, i => Foo (i));
```

или еще проще:

```
Parallel.For (0, 100, Foo);
```

А представленный далее последовательный цикл foreach:

```
foreach (char c in "Hello, world")
    Foo (c);
```

распараллеливается следующим образом:

```
Parallel.ForEach ("Hello, world", Foo);
```

Рассмотрим практический пример. Если мы импортируем пространство имен System.Security.Cryptography, то сможем генерировать шесть строк с парами открытого и секретного ключей параллельно:

```
var keyPairs = new string[6];
Parallel.For (0, keyPairs.Length,
    i => keyPairs[i] = RSA.Create().ToString (true));
```

Как и в случае Parallel.Invoke, методам Parallel.For и Parallel.ForEach можно передавать большое количество элементов работы и они будут эффективно распределены по нескольким задачам.



Последний запрос можно также построить с помощью PLINQ:

```
string[] keyPairs =
    ParallelEnumerable.Range (0, 6)
        .Select (i => RSA.Create().ToString (true))
        .ToArray();
```

Сравнение внешних и внутренних циклов

Методы Parallel.For и Parallel.ForEach обычно лучше всего работают на внешних, а не на внутренних циклах. Причина в том, что посредством внешних циклов вы предлагаете распараллеливать более крупные порции работы, снижая накладные расходы по управлению. Распараллеливание сразу внутрен-

них и внешних циклов обычно излишне. В следующем примере для получения ощутимой выгоды от распараллеливания внутреннего цикла обычно требуется более 100 ядер:

```
Parallel.For (0, 100, i =>
{
    Parallel.For (0, 50, j => Foo (i, j));      // Внутренний цикл лучше
});                                              // выполнять последовательно
```

Индексированная версия `Parallel.ForEach`

Временами полезно знать индекс итерации цикла. В случае последовательного цикла `foreach` это легко:

```
int i = 0;
foreach (char c in "Hello, world")
    Console.WriteLine (c.ToString() + i++);
```

Однако инкрементирование совместно используемой переменной не является безопасным к потокам в параллельном контексте. Взамен должна применяться следующая версия `ForEach`:

```
public static ParallelLoopResult ForEach<TSource> (
    IEnumerable<TSource> source, Action<TSource, ParallelLoopState, long> body)
```

Мы проигнорируем класс `ParallelLoopState` (он будет рассматриваться в следующем разделе). Пока что нас интересует третий параметр типа `long`, который отражает индекс цикла:

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
    Console.WriteLine (c.ToString() + i);
});
```

Чтобы задействовать такой прием в практическом примере, давайте вернемся к программе проверки орфографии, которую мы писали с помощью PLINQ. Следующий код загружает словарь и массив, содержащий миллион слов, для целей тестирования:

```
if (!File.Exists ("WordLookup.txt"))          // Содержит около 150 000 слов
    File.WriteAllText ("WordLookup.txt",
        await new HttpClient().GetStringAsync (
            "http://www.albahari.com/ispell/allwords.txt"));

var wordLookup = new HashSet<string> (
    File.ReadAllLines ("WordLookup.txt"),
    StringComparer.InvariantCultureIgnoreCase);

var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
    .Select (i => wordList [random.Next (0, wordList.Length)])
    .ToArray();

wordsToTest [12345] = "woozsh";                // Внесение пары
wordsToTest [23456] = "wubsie";                // орфографических ошибок
```

Мы можем выполнить проверку орфографии в массиве wordsToTest с использованием индексированной версии Parallel.ForEach:

```
var misspellings = new ConcurrentBag<Tuple<int, string>>();  
Parallel.ForEach (wordsToTest, (word, state, i) =>  
{  
    if (!wordLookup.Contains (word))  
        misspellings.Add (Tuple.Create ((int) i, word));  
});
```

Обратите внимание, что мы должны объединять результаты в безопасную к потокам коллекцию: по сравнению с применением PLINQ необходимость в таком действии считается недостатком. Преимущество перед PLINQ связано с тем, что мы избегаем использования индексированной операции запроса Select, которая менее эффективна, чем индексированная версия метода ForEach.

ParallelLoopState: раннее прекращение циклов

Поскольку тело цикла в параллельном методе For или ForEach представляет собой делегат, выйти из цикла до его полного завершения с помощью оператора break не получится. Взамен придется вызвать метод Break или Stop на объекте ParallelLoopState:

```
public class ParallelLoopState  
{  
    public void Break();  
    public void Stop();  
  
    public bool IsExceptional { get; }  
    public bool IsStopped { get; }  
    public long? LowestBreakIteration { get; }  
    public bool ShouldExitCurrentIteration { get; }  
}
```

Получить объект ParallelLoopState довольно просто: все версии методов For и ForEach перегружены для приема тела цикла типа Action<TSource, ParallelLoopState>. Таким образом, для распараллеливания следующего цикла:

```
foreach (char c in "Hello, world")  
    if (c == ',')  
        break;  
    else  
        Console.Write (c);
```

нужно поступить так:

```
Parallel.ForEach ("Hello, world", (c, loopState) =>  
{  
    if (c == ',')  
        loopState.Break();  
    else  
        Console.Write (c);  
});
```

Вот вывод:

Hlloe

В выводе несложно заметить, что тела циклов могут завершаться в произвольном порядке. Помимо такого отличия вызов `Break` выдает, *по меньшей мере*, те же самые элементы, что и при последовательном выполнении цикла: приведенный пример будет всегда выводить *минимум* буквы H, e, l, l и o в каком-нибудь порядке. Напротив, вызов `Stop` вместо `Break` приводит к принудительному завершению всех потоков сразу после их текущей итерации. В данном примере вызов `Stop` может дать подмножество букв H, e, l, l и o, если другой поток отстал. Вызов `Stop` полезен, когда обнаружено то, что требовалось найти, или выяснилось, что что-то пошло не так, и потому результаты просматриваться не будут.



Методы `Parallel.For` и `Parallel.ForEach` возвращают объект `ParallelLoopResult`, который открывает доступ к свойствам с именами `IsCompleted` и `LowestBreakIteration`. Они сообщают, полностью ли завершился цикл, и если это не так, то указывают, на какой итерации он был прекращен.

Если свойство `LowestBreakIteration` возвращает `null`, тогда в цикле вызывался метод `Stop` (а не `Break`).

В случае длинного тела цикла может потребоваться прервать другие потоки где-то на полпути выполнения тела метода при раннем вызове `Break` или `Stop`, что реализуется за счет опроса свойства `ShouldExitCurrentIteration` в различных местах кода. Указанное свойство принимает значение `true` немедленно после вызова `Stop` или очень скоро после вызова `Break`.



Свойство `ShouldExitCurrentIteration` также становится равным `true` после запроса отмены либо в случае генерации исключения в цикле.

Свойство `IsExceptional` позволяет узнать, произошло ли исключение в другом потоке. Любое необработанное исключение приведет к останову цикла после текущей итерации каждого потока: чтобы избежать его, вы должны явно обрабатывать исключения в своем коде.

Оптимизация посредством локальных значений

Методы `Parallel.For` и `Parallel.ForEach` предлагают набор перегруженных версий, которые работают с аргументом обобщенного типа по имени `TLocal`. Такие перегруженные версии призваны помочь оптимизировать объединение данных из циклов с интенсивными итерациями. Простейшая из них выглядит следующим образом:

```
public static ParallelLoopResult For <TLocal> (
    int fromInclusive,
    int toExclusive,
```

```
Func <TLocal> localInit,  
Func <int, ParallelLoopState, TLocal, TLocal> body,  
Action <TLocal> localFinally);
```

На практике данные методы редко востребованы, т.к. их целевые сценарии в основном покрываются PLINQ (что, в принципе, хорошо, поскольку иногда их перегруженные версии выглядят слегка устрашающими).

По существу вот в чем заключается проблема: предположим, что необходимо просуммировать квадратные корни чисел от 1 до 10 000 000. Вычисление 10 млн квадратных корней легко распараллеливается, но суммирование их значений — дело хлопотное, т.к. обновление итоговой суммы должно быть помещено внутрь блокировки:

```
object locker = new object();  
double total = 0;  
Parallel.For (1, 10000000,  
    i => { lock (locker) total += Math.Sqrt (i); });
```

Выигрыш от распараллеливания более чем нивелируется ценой получения 10 млн блокировок, а также блокированием результата.

Однако в реальности нам *не нужны* 10 млн блокировок. Представьте себе команду волонтеров по сборке большого объема мусора. Если все работники совместно пользуются единственным мусорным ведром, то хождения к нему и состязания сделают процесс крайне неэффективным. Очевидное решение предусматривает снабжение каждого работника собственным или “локальным” мусорным ведром, которое время от времени опустошается в главный контейнер.

Именно таким способом работают версии `TLocal` методов `For` и `ForEach`. Волонтеры — это внутренние рабочие потоки, а *локальное значение* представляет локальное мусорное ведро. Чтобы класс `Parallel` справился с этой работой, необходимо предоставить два дополнительных делегата.

Делегат, который указывает, каким образом инициализировать новое локальное значение.

Делегат, который указывает, каким образом объединять локальную агрегацию с главным значением.

Кроме того, вместо возвращения `void` делегат тела цикла должен возвращать новую агрегацию для локального значения. Ниже приведен переделанный пример:

```
object locker = new object();  
double grandTotal = 0;  
  
Parallel.For (1, 10000000,  
    () => 0.0,                                // Инициализировать локальное значение  
    (i, state, localTotal) =>                // Делегат тела цикла. Обратите внимание,  
    localTotal + Math.Sqrt (i),      // что он возвращает новый локальный итог  
    localTotal =>                            // Добавить локальное значение  
    { lock (locker) grandTotal += localTotal; } // к главному значению  
);
```

Здесь по-прежнему требуется блокировка, но только вокруг агрегирования локального значения с общей суммой, что делает процесс гораздо эффективнее.



Как утверждалось ранее, PLINQ часто хорошо подходит для таких сценариев. Распараллелить наш пример с помощью PLINQ можно было бы так:

```
ParallelEnumerable.Range(1, 10000000)  
    .Sum(i => Math.Sqrt(i))
```

(Обратите внимание, что мы применяем `ParallelEnumerable` для обеспечения *разбиения на основе диапазонов*: в данном случае оно улучшает производительность, потому что все числа требуют равного времени на обработку.)

В более сложных сценариях вместо `Sum` может использоваться LINQ-операция `Aggregate`. Если вы предоставите фабрику локальных начальных значений, то ситуация будет в чем-то аналогична представлению локальных значений для `Parallel.ForEach`.

Параллелизм задач

Параллелизм задач — это подход самого низкого уровня к распараллеливанию с применением инфраструктуры PFX. Классы для работы на таком уровне определены в пространстве имен `System.Threading.Tasks` и включают перечисленные ниже.

Класс	Назначение
<code>Task</code>	Для управления единицей работы
<code>Task<TResult></code>	Для управления единицей работы с возвращаемым значением
<code>TaskFactory</code>	Для создания задач
<code>TaskFactory<TResult></code>	Для создания задач и продолжений с тем же самым возвращаемым типом
<code>TaskScheduler</code>	Для управления планированием задач
<code>TaskCompletionSource</code>	Для ручного управления рабочим потоком действий задачи

Основы задач были раскрыты в главе 14; в настоящем разделе мы рассмотрим расширенные возможности задач, которые ориентированы на параллельное программирование. В частности, будут обсуждаться следующие темы:

- тонкая настройка планирования задачи;
- установка отношения “родительская/дочерняя”, когда одна задача запускается из другой;
- расширенное использование продолжений;
- класс `TaskFactory`.



Библиотека параллельных задач (TPL) позволяет создавать сотни (или даже тысячи) задач с минимальными накладными расходами. Но если необходимо создавать миллионы задач, то для поддержания эффективности задачи понадобится организовывать в более крупные единицы работы. Класс Parallel и PLINQ делают это автоматически.



В Visual Studio предусмотрено окно для мониторинга задач (Debug⇒Window⇒Parallel Tasks (Отладка⇒Окно⇒Параллельные задачи)). Окно Parallel Tasks (Параллельные задачи) эквивалентно окну Threads (Потоки), но предназначено для задач. Окно Parallel Stacks (Параллельные стеки) также поддерживает специальный режим для задач.

Создание и запуск задач

Как было описано в главе 14, метод Task.Run создает и запускает объект Task или Task<TResult>. На самом деле Task.Run является сокращением для вызова метода Task.Factory.StartNew, который предлагает более высокую гибкость через дополнительные перегруженные версии.

Указание объекта состояния

Метод Task.Factory.StartNew позволяет указывать объект *состояния*, который передается целевому методу. Сигнатура целевого метода должна в данном случае содержать одиночный параметр типа object:

```
var task = Task.Factory.StartNew (Greet, "Hello");
task.Wait(); // Ожидать, пока задача завершится.

void Greet (object state) { Console.Write (state); } // Hello
```

Такой прием дает возможность избежать затрат на замыкание, требуемое для выполнения лямбда-выражения, которое вызывает метод Greet. Это является микрооптимизацией и редко необходимо на практике, так что мы можем оставить объект состояния для более полезного сценария — назначение задаче значащего имени. Затем для запрашивания имени можно применять свойство AsyncState:

```
var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
Console.WriteLine (task.AsyncState); // Greeting
task.Wait();

void Greet (string message) { Console.Write (message); }
```



Среда Visual Studio отображает значение свойства AsyncState каждой задачи в окне Parallel Tasks, так что значащее имя задачи может основательно упростить отладку.

TaskCreationOptions

Настроить выполнение задачи можно за счет указания перечисления `TaskCreationOptions` при вызове `StartNew` (или создании объекта `Task`). `TaskCreationOptions` — перечисление флагов со следующими (комбинируемыми) значениями:

```
LongRunning, PreferFairness, AttachedToParent
```

Значение `LongRunning` предлагает планировщику выделить для задачи поток; как было показано в главе 14, это полезно для задач с интенсивным вводом-выводом, а также для длительно выполняющихся задач, которые иначе могут заставить кратко выполняющиеся задачи ожидать чрезмерно долгое время перед тем, как они будут запланированы.

Значение `PreferFairness` сообщает планировщику о необходимости попытаться обеспечить планирование задач в том порядке, в каком они были запущены. Обычно планировщик может поступать иначе, потому что он внутренне оптимизирует планирование задач с использованием локальных очередей захвата работ — оптимизация, позволяющая создавать *дочерние* задачи без накладных расходов на состязания, которые в противном случае возникли бы при доступе к единственной очереди работ. Дочерняя задача создается путем указания значения `AttachedToParent`.

Дочерние задачи

Когда одна задача запускает другую, можно дополнительно установить отношение “родительская/дочерняя”:

```
Task parent = Task.Factory.StartNew () =>
{
    Console.WriteLine ("I am a parent");
    Task.Factory.StartNew () => // Отсоединенная задача
    {
        Console.WriteLine ("I am detached");
    });
    Task.Factory.StartNew () => // Дочерняя задача
    {
        Console.WriteLine ("I am a child");
    }, TaskCreationOptions.AttachedToParent);
});
```

Дочерняя задача специфична тем, что при ожидании завершения *родительской* задачи ожидаются также и любые ее дочерние задачи. До этой точки поднимаются любые дочерние исключения:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew () =>
{
    Task.Factory.StartNew () => // Дочерняя
    {
        Task.Factory.StartNew () => { throw null; }, atp); // Внучатая
    }, atp);
}); // Следующий вызов генерирует исключение NullReferenceException
// (помещенное в оболочку AggregateExceptions):
parent.Wait();
```

Как вскоре будет показано, прием может оказаться особенно полезным, когда дочерняя задача является продолжением.

Ожидание на множестве задач

В главе 14 упоминалось, что организовать ожидание на одиночной задаче можно либо вызовом ее метода `Wait`, либо обращением к ее свойству `Result` (в случае `Task<TResult>`). Можно также реализовать ожидание сразу на множестве задач — с помощью статических методов `Task.WaitAll` (ожидание завершения всех указанных задач) и `Task.WaitAny` (ожидание завершения какой-то одной задачи).

Метод `WaitAll` похож на ожидание каждой задачи по очереди, но более эффективен тем, что требует (максимум) одного переключения контекста. Кроме того, если в одной или большем количестве задач генерируется необработанное исключение, то `WaitAll` по-прежнему ожидает каждую задачу. Затем он повторно генерирует исключение `AggregateException`, в котором накоплены исключения из всех отказавших задач (ситуация, когда класс `AggregateException` по-настоящему полезен). Ниже показан эквивалентный код:

```
// Предполагается, что t1, t2 и t3 - задачи:  
var exceptions = new List<Exception>();  
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }  
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }  
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }  
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

Вызов метода `WaitAny` эквивалентен ожиданию события `ManualResetEventSlim`, которое сигнализируется каждой задачей, как только она завершена.

Помимо времени тайм-аута методам `Wait` можно также передавать признак отмены: это позволяет отменить ожидание, но не саму задачу.

Отмена задач

При запуске задачи можно дополнительно передавать признак отмены. Если позже через данный признак произойдет отмена, то задача войдет в состояние `Canceled` (отменена):

```
var cts = new CancellationTokenSource();  
CancellationToken token = cts.Token;  
cts.CancelAfter (500);  
  
Task task = Task.Factory.StartNew () =>  
{  
    Thread.Sleep (1000);  
    token.ThrowIfCancellationRequested(); // Проверить запрос отмены  
, token);  
    try { task.Wait(); }  
    catch (AggregateException ex)  
{  
        Console.WriteLine (ex.InnerException is TaskCanceledException); // True  
        Console.WriteLine (task.IsCanceled); // True  
        Console.WriteLine (task.Status); // Canceled  
    }  
}
```

`TaskCanceledException` — подкласс класса `OperationCanceledException`. Если нужно явно сгенерировать исключение `OperationCanceledException` (вместо вызова `token.ThrowIfCancellationRequested`), тогда потребуется передать признак отмены конструктору `OperationCanceledException`. Если это не сделано, то задача не войдет в состояние `TaskStatus.Canceled` и не будет инициировать продолжение `OnlyOnCanceled`.

Если задача отменяется еще до своего запуска, тогда она не будет запланирована — в таком случае исключение `OperationCanceledException` генерируется немедленно.

Поскольку признаки отмены распознаются другими API-интерфейсами, их можно передавать другим конструкциям и отмена будет распространяться гладким образом:

```
var cancelSource = new CancellationSource();
CancellationToken token = cancelSource.Token;
Task task = Task.Factory.StartNew (() =>
{
    // Передать признак отмены в запрос PLINQ:
    var query = someSequence.AsParallel().WithCancellation (token)...
    ...выполнить перечисление результатов запроса...
});
```

Вызов `Cancel` на `cancelSource` в рассмотренном примере приведет к отмене запроса PLINQ с генерацией исключения `OperationCanceledException` в теле задачи, которое затем отменит задачу.



Признаки отмены, которые можно передавать в методы, подобные `Wait` и `CancelAndWait`, позволяют отменить операцию ожидания, а не саму задачу.

Продолжение

Метод `ContinueWith` выполняет делегат немедленно после завершения задачи:

```
// предшественник
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedent..."));
// продолжение
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

Как только задача `task1` (*предшественник*) завершается, отказывает или отменяется, запускается задача `task2` (*продолжение*). (Если задача `task1` была завершена до того, как выполнилась вторая строка кода, то задача `task2` будет запланирована для незамедлительного выполнения.) Аргумент `ant`, переданный лямбда-выражению продолжения, представляет собой ссылку на предшествующую задачу. Сам метод `ContinueWith` возвращает задачу, облегчая добавление дополнительных продолжений.

По умолчанию предшествующая задача и задача продолжения могут выполняться в разных потоках. Указав `TaskContinuationOptions.ExecuteSynchronously` при вызове `ContinueWith`, можно заставить их выполнятся в одном и том же потоке: это позволяет улучшить производительность при мелкомодульных продолжениях за счет уменьшения косвенности.

Продолжение и Task<TResult>

Подобно обычным задачам продолжения могут иметь тип Task<TResult> и возвращать данные. В следующем примере мы вычисляем Math.Sqrt(8*2) с применением последовательности соединенных в цепочку задач и выводим результат:

```
Task.Factory.StartNew<int> (() => 8)
    .ContinueWith (ant => ant.Result * 2)
    .ContinueWith (ant => Math.Sqrt (ant.Result))
    .ContinueWith (ant => Console.WriteLine (ant.Result)); // 4
```

Из-за стремления к простоте этот пример получился несколько неестественным; в реальных проектах такие лямбда-выражения могли бы вызывать функции с интенсивными вычислениями.

Продолжение и исключения

Продолжение может узнать, отказал ли предшественник, запросив свойство Exception предшествующей задачи или просто вызвав метод Result/Wait и перехватив результирующее исключение AggregateException. Если предшественник отказал, и то же самое сделало продолжение, тогда исключение считается *необнаруженным*; в таком случае при последующей обработке задачи сборщиком мусора будет сгенерировано статическое исключение TaskScheduler.UnobservedTaskException.

Безопасный шаблон предполагает повторную генерацию исключений предшественника. До тех пор пока ожидается продолжение, исключение будет распространяться и повторно генерироваться для ожидающей задачи:

```
Task continuation = Task.Factory.StartNew ((() => { throw null; }))
    .ContinueWith (ant =>
{
    ant.Wait();
    // Продолжить обработку...
});

continuation.Wait(); // Исключение теперь передается обратно вызывающей задаче
```

Еще один способ работы с исключениями предусматривает указание разных продолжений для исходов с исключениями и без исключений, что делается с помощью перечисления TaskContinuationOptions:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });

Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                TaskContinuationOptions.OnlyOnFaulted);

Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                            TaskContinuationOptions.NotOnFaulted);
```

Как вскоре будет показано, такой шаблон особенно удобен в сочетании с дочерними задачами.

Следующий расширяющий метод “поглощает” необработанные исключения задачи:

```

public static void IgnoreExceptions (this Task task)
{
    task.ContinueWith (t => { var ignore = t.Exception; },
        TaskContinuationOptions.OnlyOnFaulted);
}

```

(Метод может быть улучшен добавлением кода для регистрации исключений.) Вот как его можно использовать:

```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

Продолжение и дочерние задачи

Мощная особенность продолжений связана с тем, что они запускаются, только когда завершены все дочерние задачи (рис. 22.5). В этой точке любые исключения, сгенерированные дочерними задачами, маршируются в продолжение.

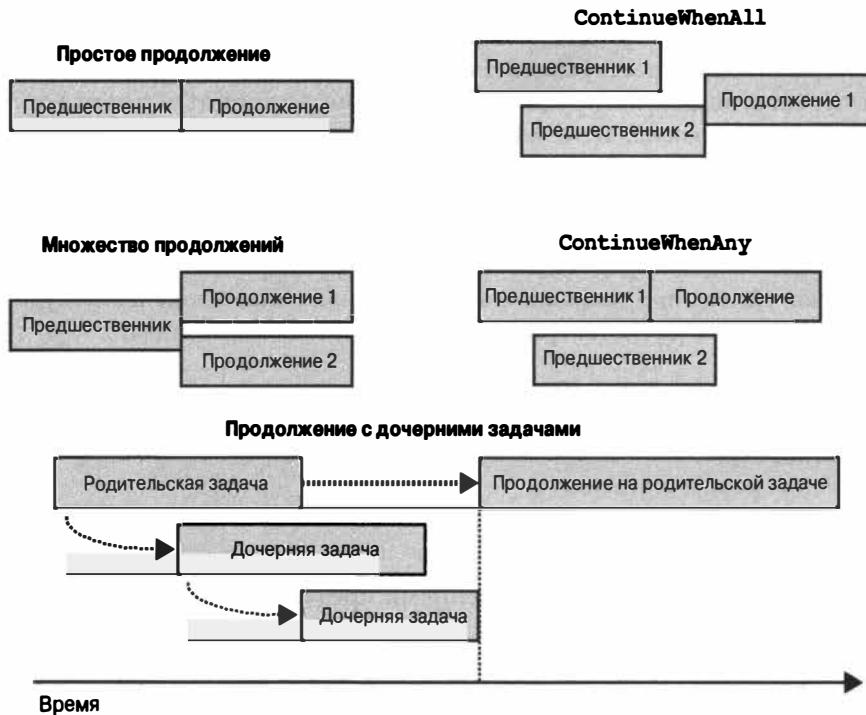


Рис. 22.5. Продолжения

В следующем примере мы начинаем три дочерние задачи, каждая из которых генерирует исключение `NullReferenceException`. Затем мы перехватываем все исключения сразу через продолжение на родительской задаче:

```

TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
    Task.Factory.StartNew (() => { throw null; }, atp);
    Task.Factory.StartNew (() => { throw null; }, atp);
    Task.Factory.StartNew (() => { throw null; }, atp);
}

```

```
    Task.Factory.StartNew(() => { throw null; }, atp);
})
.ContinueWith(p => Console.WriteLine(p.Exception),
    TaskContinuationOptions.OnlyOnFaulted);
```

Условные продолжения

По умолчанию продолжение планируется *безусловным образом* — независимо от того, завершена предшествующая задача, сгенерировано исключение или задача была отменена. Такое поведение можно изменить с помощью набора (комбинируемых) флагов, определенных в перечислении `TaskContinuationOptions`. Вот три основных флага, которые управляют условным продолжением:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

Флаги являются субтрактивными в том смысле, что чем больше их применяется, тем менее вероятно выполнение продолжения. Для удобства также представляются следующие заранее скомбинированные значения:

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Объединение всех флагов `Not*`, т.е. `NotOnRanToCompletion`, `NotOnFaulted` и `NotOnCanceled`, бессмысленно, т.к. в результате продолжение будет всегда отменяться.)

Наличие “`RanToCompletion`” в имени означает успешное завершение предшествующей задачи — без отмены или необработанных исключений.

Наличие “`Faulted`” в имени означает, что в предшествующей задаче было сгенерировано необработанное исключение.

Наличие “`Canceled`” в имени означает одну из следующих двух ситуаций.

- Предшествующая задача была отменена через ее признак отмены. Другими словами, в предшествующей задаче было сгенерировано исключение `OperationCanceledException`, свойство `CancellationToken` которого соответствует тому, что передавалось предшествующей задаче во время запуска.
- Предшествующая задача была неявно отменена, поскольку она не удовлетворила предикат условного продолжения.

Важно понимать, что когда продолжение не выполнилось из-за упомянутых флагов, оно не забыто и не отброшено — это продолжение *отменено*. Другими словами, любые продолжения на самом отмененном продолжении *затем запустятся*, если только вы не указали в условии флаг `NotOnCanceled`. Например, взгляните на приведенный далее код:

```
Task t1 = Task.Factory.StartNew(...);
Task fault = t1.ContinueWith(ant => Console.WriteLine("fault"),
    TaskContinuationOptions.OnlyOnFaulted);
Task t3 = fault.ContinueWith(ant => Console.WriteLine("t3"));
```

Несложно заметить, что задача `t3` всегда будет запланирована — даже если `t1` не генерирует исключение (рис. 22.6). Причина в том, что если задача `t1` завершена успешно, тогда задача `fault` будет отменена, и с учетом отсутствия ограничений продолжения задача `t3` будет запущена безусловным образом.



Рис. 22.6. Условные продолжения

Если нужно, чтобы задача `t3` выполнялась, только если действительно была запущена задача `fault`, то потребуется поступить так:

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),  
                               TaskContinuationOptions.NotOnCanceled);
```

(В качестве альтернативы мы могли бы указать `OnlyOnRanToCompletion`; разница в том, что тогда задача `t3` не запустилась бы в случае генерации исключения внутри задачи `fault`.)

Продолжение на основе множества предшествующих задач

С помощью методов `ContinueWhenAll` и `ContinueWhenAny` класса `TaskFactory` выполнение продолжения можно планировать на основе завершения множества предшествующих задач. Однако эти методы стали избыточными после появления комбинаторов задач, которые обсуждались в главе 14 (`WhenAll` и `WhenAny`). В частности, при наличии следующих задач:

```
var task1 = Task.Run (() => Console.Write ("X"));  
var task2 = Task.Run (() => Console.Write ("Y"));
```

вот как можно запланировать выполнение продолжения, когда обе они завершатся:

```
var continuation = Task.Factory.ContinueWhenAll (  
    new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

Тот же результат легко получить с помощью комбинатора задач `WhenAll`:

```
var continuation = Task.WhenAll (task1, task2)  
    .ContinueWith (ant => Console.WriteLine ("Done"));
```

Множество продолжений на единственной предшествующей задаче

Вызов `ContinueWith` более одного раза на той же самой задаче создает множество продолжений на единственном предшественнике. Когда предшественник завершается, все продолжения запускаются вместе (если только не было указано значение `TaskContinuationOptions.ExecuteSynchronously`, из-за чего продолжения будут выполняться последовательно).

Следующий код ожидает одну секунду, а затем выводит на консоль либо XY, либо YX:

```
var t = Task.Factory.StartNew(() => Thread.Sleep(1000));
t.ContinueWith(ant => Console.WriteLine("X"));
t.ContinueWith(ant => Console.WriteLine("Y"));
```

Планировщики задач

Планировщик задач распределяет задачи по потокам и представлен абстрактным классом `TaskScheduler`. В .NET предлагаются две конкретные реализации: *стандартный планировщик*, который работает в тандеме с пулом потоков CLR, и *планировщик контекста синхронизации*. Последний предназначен (главным образом) для содействия в работе с потоковой моделью WPF и Windows Forms, которая требует, чтобы доступ к элементам управления пользовательского интерфейса осуществлялся только из создавшего их потока (см. раздел “Многопоточность в обогащенных клиентских приложениях” в главе 14). Захватив такой планировщик, мы можем сообщить задаче или продолжению о выполнении в этом контексте:

```
// Предположим, что мы находимся в потоке пользовательского
// интерфейса внутри приложения Windows Forms или WPF:
_uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
```

Предполагая, что `Foo` — метод с интенсивными вычислениями, возвращающий строку, а `lblResult` — метка WPF или Windows Forms, вот как можно было бы безопасно обновить метку после завершения операции:

```
Task.Run(() => Foo())
    .ContinueWith(ant => lblResult.Content = ant.Result, _uiScheduler);
```

Разумеется, для действий подобного рода чаще будут использоваться асинхронные функции C#.

Возможно также написание собственного планировщика задач (путем создания подкласса `TaskScheduler`), хотя это делается только в очень специализированных сценариях. Для специального планирования чаще всего будет применяться класс `TaskCompletionSource`.

TaskFactory

Когда вызывается `Task.Factory`, происходит обращение к статическому свойству класса `Task`, которое возвращает стандартный объект фабрики задач, т.е. `TaskFactory`. Назначение фабрики задач заключается в создании задач — в частности трех их видов:

- “обыкновенных” задач (через метод `StartNew`);
- продолжений с множеством предшественников (через методы `ContinueWhenAll` и `ContinueWhenAny`);
- задач, которые являются оболочками для методов, следующих устаревшему шаблону APM (через метод `FromAsync`; см. раздел “Устаревшие шаблоны” в главе 14).

Еще один способ создания задач предусматривает создание экземпляра Task и вызов метода Start. Тем не менее, подобным образом можно создавать только “обыкновенные” задачи, но не продолжения.

Создание собственных фабрик задач

Класс TaskFactory — не абстрактная фабрика: на самом деле вы можете создавать объекты данного класса, что удобно, когда нужно многократно создавать задачи с использованием тех же самых (нестандартных) значений для TaskCreationOptions, TaskContinuationOptions или TaskScheduler. Например, если требуется многократно создавать длительно выполняющиеся родительские задачи, тогда специальную фабрику можно построить следующим образом:

```
var factory = new TaskFactory (
    TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
    TaskContinuationOptions.None);
```

Затем создание задач сводится просто к вызову метода StartNew на фабрике:

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

Специальные опции продолжения применяются в случае вызова методов ContinueWhenAll и ContinueWhenAny.

Работа с AggregateException

Как вы уже видели, инфраструктура PLINQ, класс Parallel и объекты Task автоматически маршиализируют исключения потребителю. Чтобы понять, почему это важно, рассмотрим показанный ниже запрос LINQ, который на первой итерации генерирует исключение DivideByZeroException:

```
try
{
    var query = from i in Enumerable.Range (0, 1000000)
                select 100 / i;
    ...
}
catch (DivideByZeroException)
{
    ...
}
```

Если запросить у инфраструктуры PLINQ распараллеливание такого запроса, и она проигнорирует обработку исключений, то вполне возможно, что исключение DivideByZeroException сгенерируется в *отдельном потоке*, пропустив ваш блок catch и вызвав аварийное завершение приложения.

Поэтому исключения автоматически перехватываются и повторно генерируются для вызывающего потока. Но, к сожалению, дело не сводится просто к перехвату DivideByZeroException. Поскольку параллельные библиотеки задействуют множество потоков, вполне возможна одновременная генера-

ция двух и более исключений. Чтобы обеспечить получение сведений обо всех исключениях, по указанной причине исключения помещаются в контейнер AggregateException, свойство InnerExceptions которого содержит каждое из перехваченных исключений:

```
try
{
    var query = from i in ParallelEnumerable.Range (0, 1000000)
                select 100 / i;
    // Выполнить перечисление результатов запроса
    ...
}
catch (AggregateException aex)
{
    foreach (Exception ex in aex.InnerExceptions)
        Console.WriteLine (ex.Message);
}
```



Как инфраструктура PLINQ, так и класс Parallel при обнаружении первого исключения заканчивают выполнение запроса или цикла, не обрабатывая любые последующие элементы либо итерации тела цикла. Однако до завершения текущей итерации цикла могут быть сгенерированы дополнительные исключения. Первое возникшее исключение в AggregateException доступно через свойство InnerException.

Flatten и Handle

Класс AggregateException предоставляет пару методов для упрощения обработки исключений: Flatten и Handle.

Flatten

Объекты AggregateException довольно часто будут содержать другие объекты AggregateException. Пример, когда подобное может произойти — ситуация, при которой дочерняя задача генерирует исключение. Чтобы упростить обработку, можно устраниТЬ любой уровень вложения, вызвав метод Flatten. Этот метод возвращает новый объект AggregateException с обычным плоским списком внутренних исключений:

```
catch (AggregateException aex)
{
    foreach (Exception ex in aex.Flatten().InnerExceptions)
        myLogWriter.LogError (ex);
}
```

Handle

Иногда полезно перехватывать исключения только специфических типов, а исключения других типов генерировать повторно. Метод Handle класса

`AggregateException` предлагает удобное сокращение. Он принимает предикат исключений, который будет запускаться на каждом внутреннем исключении:

```
public void Handle (Func<Exception, bool> predicate)
```

Если предикат возвращает `true`, то считается, что исключение “обработано”. После того, как делегат запустится на всех исключениях, произойдет следующее:

- если все исключения были “обработаны” (делегат вернул `true`), то исключение не генерируется повторно;
- если были исключения, для которых делегат вернул `false` (“необработанные”), то строится новый объект `AggregateException`, содержащий такие исключения, и затем он генерируется повторно.

Например, приведенный далее код в конечном итоге повторно генерирует другой объект `AggregateException`, который содержит одиночное исключение `NullReferenceException`:

```
var parent = Task.Factory.StartNew (() =>
{
    // Мы сгенерируем 3 исключения сразу, используя 3 дочерние задачи:
    int[] numbers = { 0 };

    var childFactory = new TaskFactory
        (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);

    childFactory.StartNew (() => 5 / numbers[0]); // Деление на ноль
    childFactory.StartNew (() => numbers [1]); // Выход индекса за
                                                // допустимые пределы
    childFactory.StartNew (() => { throw null; }); // Ссылка null
});

try { parent.Wait(); }
catch (AggregateException aex)
{
    aex.Flatten().Handle (ex => // Обратите внимание, что
                                // по-прежнему нужно вызывать Flatten
    {
        if (ex is DivideByZeroException)
        {
            Console.WriteLine ("Divide by zero"); // Деление на ноль
            return true; // Это исключение "обработано"
        }
        if (ex is IndexOutOfRangeException)
        {
            Console.WriteLine ("Index out of range"); // Выход индекса за
                                                        // допустимые пределы
            return true; // Это исключение "обработано"
        }
        return false; // Все остальные исключения будут сгенерированы повторно
    });
}
```

Параллельные коллекции

.NET предлагает коллекции, безопасные в отношении потоков, которые определены в пространстве имен System.Collections.Concurrent:

Параллельная коллекция	Непараллельный эквивалент
ConcurrentStack<T>	Stack<T>
ConcurrentQueue<T>	Queue<T>
ConcurrentBag<T>	(отсутствует)
ConcurrentDictionary< TKey, TValue >	Dictionary< TKey, TValue >

Параллельные коллекции оптимизированы для сценариев с высокой степенью параллелизма; тем не менее, они также могут быть полезны в ситуациях, когда требуется коллекция, безопасная к потокам (в качестве альтернативы применению блокировки к обычной коллекции). Однако есть несколько предостережений.

- По производительности традиционные коллекции превосходят параллельные коллекции во всех сценариях кроме тех, которые характеризуются высокой степенью параллелизма.
- Безопасная к потокам коллекция вовсе не гарантирует, что код, в котором она используется, будет безопасным в отношении потоков (см. раздел “Блокирование и безопасность к потокам” в главе 21).
- Если вы производите перечисление параллельной коллекции, в то время как другой поток ее модифицирует, то никаких исключений не возникает — взамен вы получите смесь старого и нового содержимого.
- Параллельной версии List<T> не существует.
- Параллельные классы стеков, очередей и пакетов внутренне реализованы с помощью связных списков. Это делает их менее эффективными в плане потребления памяти, чем непараллельные классы Stack и Queue, но лучшими для параллельного доступа, т.к. связные списки способствуют построению реализаций с низкой блокировкой или вообще без таковой. (Причина в том, что вставка узла в связный список требует обновления лишь пары ссылок, тогда как вставка элемента в структуру, подобную List<T>, может привести к перемещению тысяч существующих элементов.)

Другими словами, параллельные коллекции не являются простыми сокращениями для применения обычных коллекций с блокировками. В качестве демонстрации, если запустить следующий код в *одиночном* потоке:

```
var d = new ConcurrentDictionary<int, int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

то он выполнится в три раза медленнее, чем такой код:

```
var d = new Dictionary<int, int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

(Тем не менее, чтение из ConcurrentDictionary будет быстрым, потому что операции чтения свободны от блокировок.)

Параллельные коллекции также отличаются от традиционных коллекций тем, что они открывают доступ к специальным методам, которые предназначены для выполнения атомарных операций типа “проверить и действовать”, подобных TryPop. Большинство таких методов унифицировано посредством интерфейса IProducerConsumerCollection<T>.

IProducerConsumerCollection<T>

Коллекция производителей/потребителей является одной из тех, для которых предусмотрены два главных сценария использования:

- добавление элемента (действие “производителя”);
- извлечение элемента с его удалением (действие “потребителя”).

Классическими примерами являются стеки и очереди. Коллекции производителей/потребителей играют важную роль в параллельном программировании, т.к. они способствуют построению эффективных реализаций, свободных от блокировок.

Интерфейс IProducerConsumerCollection<T> представляет безопасную к потокам коллекцию производителей/потребителей и реализован следующими классами:

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

Интерфейс IProducerConsumerCollection<T> расширяет ICollection, добавляя перечисленные ниже методы:

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

Методы TryAdd и TryTake проверяют, может ли быть выполнена операция добавления/удаления, и если может, тогда производят добавление/удаление. Проверка и действие выполняются атомарно, устранив необходимость в блокировке, к которой пришлось бы прибегнуть в случае традиционной коллекции:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

Метод TryTake возвращает false, если коллекция пуста. Метод TryAdd всегда выполняется успешно и возвращает true в предоставленных трех реализациях. Однако если вы разрабатываете собственную параллельную коллекцию, в которой дубликаты запрещены, то обеспечите возврат методом TryAdd значения false, когда заданный элемент уже существует (примером может служить реализация параллельного набора).

Конкретный элемент, который TryTake удаляет, определяется подклассом:

- в случае стека TryTake удаляет элемент, добавленный позже всех других;
- в случае очереди TryTake удаляет элемент, добавленный раньше всех других;
- в случае пакета TryTake удаляет любой элемент, который может быть удален наиболее эффективно.

Три конкретных класса главным образом реализуют методы TryTake и TryAdd явно, делая доступной ту же самую функциональность через открытые методы с более специфичными именами, такими как TryDequeue и TryPop.

ConcurrentBag<T>

Класс ConcurrentBag<T> хранит *неупорядоченную* коллекцию объектов (с разрешенными дубликатами). Класс ConcurrentBag<T> подходит в ситуациях, когда не имеет значения, какой элемент будет получен при вызове Take или TryTake.

Преимущество ConcurrentBag<T> перед параллельной очередью или стеком связано с тем, что метод Add пакета не допускает почти никаких состязаний, когда вызывается многими потоками одновременно. В отличие от него вызов Add параллельно на очереди или стеке приводит к некоторым состязаниям (хотя и намного меньшим, чем при блокировании *непараллельной* коллекции). Вызов Take на параллельном пакете тоже очень эффективен — до тех пор, пока каждый поток не извлекает большее количество элементов, чем он добавил с помощью Add.

Внутри параллельного пакета каждый поток получает свой закрытый связанный список. Элементы добавляются в закрытый список, который принадлежит потоку, вызывающему Add, что устраняет состязания. Когда производится перечисление пакета, перечислитель проходит по закрытым спискам всех потоков, выдавая каждый из их элементов по очереди.

Когда вызывается метод Take, пакет сначала просматривает закрытый список текущего потока. Если в нем имеется хотя бы один элемент¹, то задача может быть завершена легко и без состязаний. Но если этот список пуст, то пакет должен “позаимствовать” элемент из закрытого списка другого потока, что потенциально может привести к состязаниям.

Таким образом, чтобы соблюсти точность, вызов Take дает элемент, который был добавлен позже других в данном потоке; если же в этом потоке элементов нет, тогда Take дает последний добавленный элемент в другом потоке, выбранном произвольно.

Параллельные пакеты идеальны, когда параллельная операция на коллекции в основном состоит из добавления элементов посредством Add — или когда количество вызовов Add и Take сбалансировано в рамках потока. Пример первой ситуации приводился ранее во время применения метода Parallel.ForEach при реализации параллельной программы проверки орфографии:

¹ Из-за деталей реализации на самом деле должны существовать хотя бы два элемента, чтобы полностью избежать состязаний.

```
var misspellings = new ConcurrentBag<Tuple<int, string>>();  
Parallel.ForEach (wordsToTest, (word, state, i) =>  
{  
    if (!wordLookup.Contains (word))  
        misspellings.Add (Tuple.Create ((int) i, word));  
});
```

Параллельный пакет может оказаться неудачным выбором для очереди производителей/потребителей, поскольку элементы добавляются и удаляются *разными* потоками.

BlockingCollection<T>

В случае вызова метода TryTake на любой коллекции производителей/потребителей, рассмотренной в предыдущем разделе, т.е. ConcurrentStack<T>, ConcurrentQueue<T> и ConcurrentBag<T>, он возвращает false, если коллекция пуста. Иногда в таком сценарии полезнее организовать ожидание, пока элемент не станет доступным.

Вместо перегрузки методов TryTake для обеспечения такой функциональности (что привело бы к перенасыщению членами после предоставления возможности работы с признаками отмены и тайм-аутами) проектировщики PFX инкапсулировали ее в класс-оболочку по имени BlockingCollection<T>. Блокирующая коллекция может содержать внутри любую коллекцию, которая реализует интерфейс IProducerConsumerCollection<T>, и позволяет получать с помощью метода Take элемент из внутренней коллекции, обеспечивая блокирование, когда доступных элементов нет.

Блокирующая коллекция также позволяет ограничивать общий размер коллекции, блокируя *производителя*, если этот размер превышен. Коллекция, ограниченная в подобной манере, называется *ограниченной блокирующей коллекцией*.

Для использования класса BlockingCollection<T> необходимо выполнить описанные ниже шаги.

1. Создать экземпляр класса, дополнительно указывая помещаемую внутрь реализацию IProducerConsumerCollection<T> и максимальный размер (границу) коллекции.
2. Вызывать метод Add или TryAdd для добавления элементов во внутреннюю коллекцию.
3. Вызывать метод Take или TryTake для удаления (потребления) элементов из внутренней коллекции.

Если конструктор вызван без передачи ему коллекции, то автоматически будет создан экземпляр ConcurrentQueue<T>. Методы производителя и потребителя позволяют указывать признаки отмены и тайм-ауты. Методы Add и TryAdd могут блокироваться, если размер коллекции ограничен; методы Take и TryTake блокируются на время, пока коллекция пуста.

Еще один способ потребления элементов предполагает вызов метода GetConsumingEnumerable. Он возвращает (потенциально) бесконечную по-

следовательность, которая выдает элементы по мере того, как они становятся доступными. Чтобы принудительно завершить такую последовательность, необходимо вызвать `CompleteAdding`: этот метод также предотвращает дальнейшее помещение в очередь элементов.

Кроме того, класс `BlockingCollection` предоставляет статические методы под названиями `AddToAny` и `TakeFromAny`, которые позволяют добавлять и получать элемент, указывая несколько блокирующих коллекций. Действие затем будет выполнено первой коллекцией, которая способна обслужить данный запрос.

Реализация очереди производителей/потребителей

Очередь производителей/потребителей — структура, полезная как при параллельном программировании, так и в общих сценариях параллелизма. Ниже описаны основные аспекты ее работы.

- Очередь настраивается для описания элементов работы или данных, над которыми выполняется работа.
- Когда задача должна выполниться, она ставится в очередь, а вызывающий код занимается другой работой.
- Один или большее число рабочих потоков функционируют в фоновом режиме, извлекая и запуская элементы из очереди.

Очередь производителей/потребителей обеспечивает точный контроль над тем, сколько рабочих потоков выполняется за раз, что полезно для ограничения эксплуатации не только ЦП, но также и других ресурсов. Скажем, если задачи выполняют интенсивные операции дискового ввода-вывода, то можно ограничить параллелизм, не истощая операционную систему и другие приложения. На протяжении времени жизни очереди можно также динамически добавлять и удалять рабочие потоки. Пул потоков CLR сам представляет собой разновидность очереди производителей/потребителей, которая оптимизирована для кратко выполняющихся заданий с интенсивными вычислениями.

Очередь производителей/потребителей обычно хранит элементы данных, на которых выполняется (одна и та же) задача. Например, элементами данных могут быть имена файлов, а задача может осуществлять шифрование содержимого таких файлов. С другой стороны, применяя делегаты в качестве элементов, можно построить более универсальную очередь производителей/потребителей, где каждый элемент способен делать все что угодно.

В статье “*Parallel Programming*” (“Параллельное программирование”) по ссылке <http://albahari.com/threading/> мы показываем, как реализовать очередь производителей/потребителей с нуля, используя событие `AutoResetEvent` (а также впоследствии методы `Wait` и `Pulse` класса `Monitor`). Тем не менее, написание очереди производителей/потребителей с нуля стало необязательным, т.к. большая часть функциональности предлагается классом `BlockingCollection<T>`. Вот как его задействовать:

```

public class PCQueue : IDisposable
{
    BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();
    public PCQueue (int workerCount)
    {
        // Создать и запустить отдельный объект Task для каждого потребителя:
        for (int i = 0; i < workerCount; i++)
            Task.Factory.StartNew (Consume);
    }

    public void Enqueue (Action action) { _taskQ.Add (action); }

    void Consume()
    {
        // Эта последовательность, которую мы перечисляем, будет блокироваться,
        // когда нет доступных элементов, и заканчиваться, когда вызван
        // метод CompleteAdding
        foreach (Action action in _taskQ.GetConsumingEnumerable())
            action(); // Выполнить задачу.
    }

    public void Dispose() { _taskQ.CompleteAdding(); }
}

```

Поскольку конструктору `BlockingCollection` ничего не передается, он автоматически создает параллельную очередь. Если бы ему был передан объект `ConcurrentStack`, тогда мы получили бы в итоге стек производителей/потребителей.

Использование задач

Только что написанная очередь производителей/потребителей не является гибкой, т.к. мы не можем отслеживать элементы работы после их помещения в очередь. Очень полезными были бы следующие возможности:

- знать, когда элемент работы завершается (и ожидать его посредством `await`);
- отменять элемент работы;
- элегантно обрабатывать любые исключения, которые сгенерированы тем или иным элементом работы.

Идеальное решение предусматривало бы возможность возвращения методом `Enqueue` какого-то объекта, снабжающего нас описанной выше функциональностью. К счастью, уже существует класс, делающий в точности то, что нам нужно — это `Task`, объект которого можно либо сгенерировать с помощью `TaskCompletionSource`, либо создать напрямую (получив незапущенную или *холодную* задачу):

```

public class PCQueue : IDisposable
{
    BlockingCollection<Task> _taskQ = new BlockingCollection<Task>();
    public PCQueue (int workerCount)
    {

```

```

// Создать и запустить отдельный объект Task для каждого потребителя:
for (int i = 0; i < workerCount; i++)
    Task.Factory.StartNew (Consume);
}

public Task Enqueue (Action action, CancellationToken cancelToken
                     = default (CancellationToken))
{
    var task = new Task (action, cancelToken);
    _taskQ.Add (task);
    return task;
}

public Task<TResult> Enqueue<TResult> (Func<TResult> func,
                                         CancellationToken cancelToken = default (CancellationToken))
{
    var task = new Task<TResult> (func, cancelToken);
    _taskQ.Add (task);
    return task;
}

void Consume()
{
    foreach (var task in _taskQ.GetConsumingEnumerable())
        try
        {
            if (!task.IsCanceled) task.RunSynchronously();
        }
        catch (InvalidOperationException) { } // Условие состязаний
    }

    public void Dispose() { _taskQ.CompleteAdding(); }
}

```

В методе `Enqueue` мы помещаем в очередь и возвращаем вызывающему коду задачу, которая создана, но не запущена.

В методе `Consume` мы запускаем эту задачу синхронно в потоке потребителя. Мы перехватываем исключение `InvalidOperationException`, чтобы обработать маловероятную ситуацию, когда задача будет отменена в промежутке между проверкой, не отменена ли она, и ее запуском.

Ниже показано, как можно применять класс `PCQueue`:

```

var pcQ = new PCQueue (2); // Максимальная степень параллелизма равна 2
string result = await pcQ.Enqueue (() => "That was easy!");
...

```

Следовательно, мы имеем все преимущества задач — распространение исключений, возвращаемые значения и возможность отмены — и в то же время обладаем полным контролем над их планированием.



Span<T> и Memory<T>

Структуры `Span<T>` и `Memory<T>` действуют как низкоуровневые фасады для массива, строки или любого смежного блока управляемой либо неуправляемой памяти. Их главная цель — содействовать определенным видам микрооптимизации. В частности, они помогают писать код с *низким выделением памяти*, в котором выделение управляемой памяти сводится к минимуму (сокращая нагрузку на сборщик мусора), и нет необходимости в дублировании кода для разных типов входных данных. Они также делают возможным *нарезание* — работу с порциями массива, строки или блока памяти без создания копии.

Структуры `Span<T>` и `Memory<T>` особенно полезны в “горячих” точках, критичных к производительности, таких как конвейер обработки ASP.NET Core или средство разбора JSON, которое обслуживает объектную базу данных.



В случае если вы встретили такие типы в каком-то API-интерфейсе и не нуждаетесь в предлагаемом ими потенциальном выигрыше в плане производительности, тогда можете легко обойтись без них, как описано ниже.

- При вызове метода, который ожидает тип `Span<T>`, `ReadOnlySpan<T>`, `Memory<T>` или `ReadOnlyMemory<T>`, передавайте взамен массив, т.е. `T[]`. (Это работает благодаря неявным операциям преобразования.)
- Чтобы преобразовать промежуток/память в массив, вызывайте метод `ToArray`. А если `T` является `char`, то метод `ToString` преобразует промежуток/память в строку.

Начиная с версии C# 12, можно также использовать инициализаторы коллекций, чтобы создавать промежутки.

В частности, структура `Span<T>` решает две задачи.

- Она предоставляет общий интерфейс, подобный массиву, для управляемых массивов, строк и поддерживаемой указателем памяти. В результате у вас появляется свобода в плане работы с выделенной в стеке и неуправляемой памятью, избегая сборки мусора, и отсутствует необходимость дублировать код или возиться с указателями.
- Она делает возможным “нарезание”, т.е. открытие доступа к многократно используемым подразделам промежутка без создания копий.



Структура `Span<T>` состоит всего лишь из двух полей — указателя и длины. По этой причине она может представлять только смежные блоки памяти. (Если вам нужно работать с несмежными блоками памяти, то имеется класс `ReadOnlySequence<T>`, служащий в качестве связного списка.)

Поскольку структура `Span<T>` способна служить оболочкой для памяти, выделенной в стеке, существуют ограничения, касающиеся того, как можно хранить либо передавать экземпляры (обусловленные отчасти тем, что `Span<T>` является *ссылочной структурой*). Структура `Memory<T>` действует как промежуток без таких ограничений, но не может быть оболочкой для памяти, выделенной в стеке. Структура `Memory<T>` по-прежнему обеспечивает преимущество нарезания.

Каждая структура поставляется с эквивалентом, допускающим только чтение (`ReadOnlySpan<T>` и `ReadOnlyMemory<T>`). Помимо предотвращения неумышленного изменения аналоги, поддерживающие только чтение, дополнитель но улучшают производительность, снабжая компилятор и исполняющую среду добавочной свободой в плане оптимизации.

В самой платформе .NET (и в ASP.NET Core) указанные типы применяются для повышения эффективности ввода-вывода, взаимодействия с сетью, обработки строк и разбора данных JSON.



Способность структур `Span<T>` и `Memory<T>` выполнять нарезание массивов делает старый класс `ArraySegment<T>` избыточным. Для содействия в отказе от него предусмотрены неявные операции преобразования из `ArraySegment<T>` во все структуры промежутков/памяти, а также из `Memory<T>` и `ReadOnlyMemory<T>` в `ArraySegment<T>`.

Промежутки и нарезание

В отличие от массива, промежуток можно легко *нарезать* для представления различных подразделов одних и тех же данных, как показано на рис. 23.1.

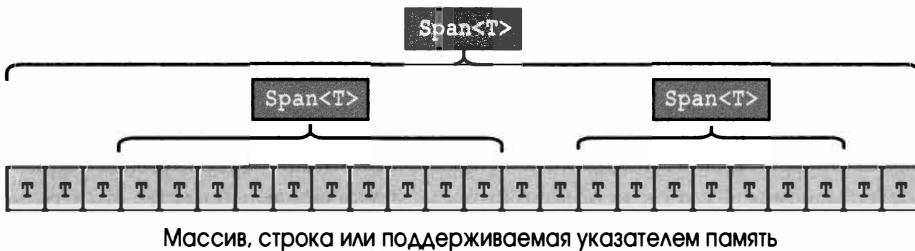


Рис. 23.1. Нарезание

Рассмотрим пример. Предположим, что вы пишете метод для суммирования элементов массива целых чисел. Реализация с микрооптимизацией позволила бы избавиться от LINQ, отдав предпочтение циклу `foreach`:

```
int Sum (int [] numbers)
{
    int total = 0;
    foreach (int i in numbers) total += i;
    return total;
}
```

Теперь представим, что вы хотите суммировать только элементы из *части* массива. У вас есть два варианта:

- сначала скопировать в другой массив ту часть массива, элементы которой необходимо просуммировать;
- добавить к методу дополнительные параметры (смещение и количество).

Первый вариант неэффективен, а второй привносит путаницу и сложность (ситуация становится еще хуже в случае методов, которые должны принимать более одного массива).

Промежутки изящно решают такую задачу. Все, что понадобится сделать — поменять тип параметра `int []` на `ReadOnlySpan<int>` (остальной код остается тем же самым):

```
int Sum (ReadOnlySpan<int> numbers)
{
    int total = 0;
    foreach (int i in numbers) total += i;
    return total;
}
```



Здесь использовался тип `ReadOnlySpan<T>`, а не `Span<T>`, т.к. модифицировать массив не нужно. Существует неявное преобразование из `Span<T>` в `ReadOnlySpan<T>`, поэтому `Span<T>` можно передавать методу, который ожидает `ReadOnlySpan<T>`.

Вот как можно протестировать метод `Sum`:

```
var numbers = new int [1000];
for (int i = 0; i < numbers.Length; i++) numbers [i] = i;
int total = Sum (numbers);
```

Метод `Sum` разрешено вызывать с массивом, потому что имеется неявное преобразование из `T []` в `Span<T>` и `ReadOnlySpan<T>`. Еще один прием предусматривает применение расширяющего метода `AsSpan`:

```
var span = numbers.AsSpan();
```

Индексатор для `ReadOnlySpan<T>` использует средство `ref readonly` языка C# для прямого доступа к лежащим в основе данным: это позволяет методу работать почти так же хорошо, как в первоначальном примере, где применялся массив. Но преимущество заключается в том, что теперь массив можно “нарезать” и суммировать только часть элементов:

```
// Суммировать 500 элементов посередине (начиная с позиции 250):  
int total = Sum (numbers.AsSpan (250, 500));
```

При наличии экземпляра `Span<T>` или `ReadOnlySpan<T>` его можно нарезать, вызывая метод `Slice`:

```
Span<int> span = numbers;  
int total = Sum (span.Slice (250, 500));
```

Можно также использовать индексы и диапазоны C# (начиная с версии C# 8):

```
Span<int> span = numbers;  
Console.WriteLine (span [1]); // Последний элемент  
Console.WriteLine (Sum (span [..10])); // Первые 10 элементов  
Console.WriteLine (Sum (span [100..])); // Элементы с 100-го и до конца  
Console.WriteLine (Sum (span [^5..])); // Последние 5 элементов
```

Хотя структура `Span<T>` не реализует интерфейс `IEnumerable<T>` (она не может реализовывать интерфейсы в силу того, что является ссылочной структурой), `Span<T>` реализует шаблон, который позволяет работать оператору `foreach` языка C# (см. раздел “Перечисление” в главе 4).

CopyTo И TryCopyTo

Метод `CopyTo` копирует элементы из одного промежутка (или `Memory<T>`) в другой. В следующем примере происходит копирование всех элементов из промежутка `x` в промежуток `y`:

```
Span<int> x = [1, 2, 3, 4]; // Выражение коллекции  
Span<int> y = new int[4];  
x.CopyTo (y);
```



Обратите внимание, что промежуток `x` был инициализирован с помощью *выражения коллекции*. Выражения коллекций (появившиеся в C# 12) — это не только удобное сокращение; в случае с промежутками они предоставляют компилятору свободу выбора лежащего в основе типа. Когда количество элементов невелико, компилятор может выделить память в стеке (вместо создания массива) и тем самым избежать накладных расходов, связанных с выделением памяти в куче.

Нарезание делает этот метод гораздо более полезным. Ниже выполняется копирование первой половины промежутка `x` во вторую половину промежутка `y`:

```
Span<int> x = [1, 2, 3, 4];  
Span<int> y = [10, 20, 30, 40];  
x[..2].CopyTo (y[2..])); // y теперь содержит [10, 20, 1, 2]
```

Если в месте назначения недостаточно пространства для завершения копирования, тогда `CopyTo` генерирует исключения, а `TryCopyTo` возвратит `false` (не копируя какие-либо элементы).

Структуры `Span<T>` и `ReadOnlySpan<T>` также открывают доступ к методам для очистки (`Clear`) и заполнения (`Fill`) промежутка, а также к методу `IndexOf` для поиска элемента в промежутке.

Поиск в промежутках

В классе `MemoryExtensions` определены многочисленные расширяющие методы, которые помогают искать значения внутри промежутков, подобные `Contains`, `IndexOf`, `LastIndexOf` и `BinarySearch` (а также методы, которые изменяют промежутки, вроде `Fill`, `Replace` и `Reverse`).

Начиная с версии .NET 8, доступны методы для поиска любого из нескольких значений, например, `ContainsAny`, `ContainsAnyExcept`, `IndexOfAny` и `IndexOfAnyExcept`. Этим методам можно указывать значения для поиска либо в виде промежутка, либо в виде экземпляра `SearchValues<T>` (из `System.Buffers`), который создается посредством вызова `SearchValues.Create`:

```
ReadOnlySpan<char> span = "The quick brown fox jumps over the lazy dog.";
var vowels = SearchValues.Create ("aeiou");
Console.WriteLine (span.IndexOfAny (vowels)); // 2
```

Класс `SearchValues<T>` позволяет повысить производительность в случае многократного использования его экземпляра в нескольких операциях поиска.



Упомянутые выше методы также можно применять при работе с массивами или строками, просто вызывая `AsSpan()` на массиве или строке.

Работа с текстом

Промежутки спроектированы для эффективной работы со строками, которые трактуются как `ReadOnlySpan<char>`. Следующий метод подсчитывает пробельные символы:

```
int CountWhitespace (ReadOnlySpan<char> s)
{
    int count = 0;
    foreach (char c in s)
        if (char.IsWhiteSpace (c))
            count++;
    return count;
}
```

Вызывать такой метод можно со строкой (благодаря неявной операции преобразования):

```
int x = CountWhitespace ("Word1 Word2"); // Нормально
```

или с подстрокой:

```
int y = CountWhitespace (someString.AsSpan (20, 10));
```

Метод `ToString` преобразует структуру `ReadOnlySpan<char>` обратно в строку.

Расширяющие методы гарантируют, что некоторые часто применяемые методы класса `string` также доступны для `ReadOnlySpan<char>`:

```
var span = "This ".AsSpan(); // ReadOnlySpan<char>
Console.WriteLine (span.StartsWith ("This")); // True
Console.WriteLine (span.Trim ().Length); // 4
```

(Обратите внимание, что по умолчанию методы вроде `StartsWith` используют *ординальное* сравнение, в то время как соответствующие методы класса `string` — сравнение, чувствительное к культуре.)

Методы наподобие `ToUpper` и `ToLower` доступны, но им придется передавать целевой промежуток с корректной длиной (это позволяет вам решить, как и где выделять память).

Некоторые методы класса `string` не будут доступными, скажем, `Split` (который расщепляет строку в массив слов). На самом деле написать прямой эквивалент метода `Split` класса `string` невозможно, поскольку нельзя создать массив промежутков.



Причина в том, что промежутки определены как *ссыпочные структуры*, которые могут существовать только в стеке.

(Под существованием только в стеке подразумевается то, что в стеке может располагаться сама структура. Содержимое, которое способен умещать промежуток, может находиться в куче — и в данном случае это так.)

Пространство имен `System.Buffers.Text` содержит дополнительные типы, помогающие работать с основанным на промежутках текстом, в том числе перечисленные ниже:

- `Utf8Formatter.TryParse` выполняет эквивалент вызова `ToString` для встроенных и простых типов, таких как `decimal`, `DateTime` и т.д., но вместо строки выдает промежуток;
- `Utf8Parser.TryParse` делает обратное и разбирает данные из промежутка в простой тип;
- `Base64` предлагает методы для чтения/записи данных Base-64.



Начиная с версии .NET 8, числовые типы и типы даты/времени .NET (а также другие простые типы) позволяют напрямую форматировать и разбирать UTF-8 с помощью новых методов `TryFormat` и `Parse/TryParse`, которые работают с типом `Span<byte>`. Новые методы определены в интерфейсах `IUtf8SpanFormattable` и `IUtf8SpanParsable<TSelf>` (последний задействует возможность C# 12 определять статические абстрактные члены интерфейса).

Фундаментальные методы CLR наподобие `int.Parse` также были перегружены с целью приема `ReadOnlySpan<char>`.

Memory<T>

Типы `Span<T>` и `ReadOnlySpan<T>` определены в виде *ссыпочных структур*, чтобы довести до максимума их оптимизационный потенциал и дать им возможность безопасно работать с памятью, выделенной в стеке (как будет

показано в следующем разделе). Однако они также накладывают ограничения. Помимо того, что ссылочные структуры недружественны к массивам, их также нельзя применять для полей в классе (в таком случае они размещались бы в куче). В свою очередь это препятствует их появлению в лямбда-выражениях, а также в качестве параметров в асинхронных методах, итераторах и асинхронных потоках данных:

```
async void Foo (Span<int> notAllowed) // Ошибка на этапе компиляции!
```

(Вспомните, что компилятор обрабатывает асинхронные методы и итераторы, реализуя закрытый *конечный автомат*, т.е. любые параметры и локальные переменные в итоге становятся полями. То же самое применимо к лямбда-выражениям, которые охватывают переменные: они будут полями в *замыканиях*.)

Структуры `Memory<T>` и `ReadOnlyMemory<T>` учитывают это и действуют в качестве промежутков, которые не могут служить оболочками для памяти, выделенной в стеке, что делает возможным их использование в полях, лямбда-выражениях, асинхронных методах и т.д.

Экземпляр `Memory<T>` или `ReadOnlyMemory<T>` можно получать из массива через неявное преобразование либо расширяющий метод `AsMemory`:

```
Memory<int> mem1 = new int[] { 1, 2, 3 };
var mem2 = new int[] { 1, 2, 3 }.AsMemory();
```

Экземпляр `Memory<T>` или `ReadOnlyMemory<T>` можно легко “преобразовать” в экземпляр `Span<T>` или `ReadOnlySpan<T>` через его свойство `Span`, так что есть возможность взаимодействовать с ним, как если бы он был промежутком. Преобразование эффективно в том, что оно не выполняет никакого копирования:

```
async void Foo (Memory<int> memory)
{
    Span<int> span = memory.Span;
    ...
}
```

(Экземпляр `Memory<T>` или `ReadOnlyMemory<T>` можно также нарезать напрямую через его метод `Slice` или диапазон C# и получать доступ к длине через свойство `Length`.)



Еще один способ получения экземпляра `Memory<T>` предусматривает его аренду из *пула* с применением класса `System.Buffers.MemoryPool<T>`. Пул памяти работает аналогично пулу массивов (см. раздел “Организация пула массивов” в главе 12) и предлагает другую стратегию для сокращения нагрузки на сборщик мусора.

В предыдущем разделе было указано, что написать прямой эквивалент метода `string.Split` для промежутков не удастся, поскольку создавать массив промежутков нельзя. К типу `ReadOnlyMemory<char>` такое ограничение неприменимо:

```
// Разбить строку на слова:
IEnumerable<ReadOnlyMemory<char>> Split (ReadOnlyMemory<char> input)
{
    int wordStart = 0;
    for (int i = 0; i <= input.Length; i++)
        if (i == input.Length || char.IsWhiteSpace (input.Span [i]))
        {
            yield return input [wordStart..i]; // Нарезать с помощью
                                              // операции диапазона C#
            wordStart = i + 1;
        }
}
```

Реализация более эффективна, чем метод `Split` класса `string`: вместо создания новых строк для каждого слова она возвращает *части* исходной строки:

```
foreach (var slice in Split ("The quick brown fox jumps over the lazy dog"))
{
    // slice - экземпляр ReadOnlyMemory<char>
}
```



Экземпляр `Memory<T>` легко преобразовать в экземпляр `Span<T>` (через свойство `Span`), но не наоборот. По этой причине при наличии выбора лучше писать методы, которые принимают тип `Span<T>`, а не `Memory<T>`.

По той же самой причине лучше писать методы, принимающие тип `ReadOnlySpan<T>`, а не `Span<T>`.

Однонаправленные перечислители

В предыдущем разделе тип `ReadOnlyMemory<char>` был задействован в качестве решения для реализации метода `Split` в строковом стиле. Но, отказавшись от `ReadOnlySpan<char>`, мы утратили возможность нарезать промежутки, поддерживаемые неуправляемой памятью. Давайте вернемся к `ReadOnlySpan<char>` и посмотрим, сможем ли мы отыскать другое решение.

Возможный вариант мог бы предусматривать написание метода `Split` в таком виде, чтобы он возвращал *диапазоны*:

```
Range[] Split (ReadOnlySpan<char> input)
{
    int pos = 0;
    var list = new List<Range>();
    for (int i = 0; i <= input.Length; i++)
        if (i == input.Length || char.IsWhiteSpace (input [i]))
        {
            list.Add (new Range (pos, i));
            pos = i + 1;
        }
    return list.ToArray();
}
```

Затем в вызывающем коде возвращенные диапазоны можно было бы использовать для нарезания исходного промежутка:

```

ReadOnlySpan<char> source = "The quick brown fox";
foreach (Range range in Split (source))
{
    ReadOnlySpan<char> wordSpan = source [range];
    ...
}

```

Это улучшение, но все еще несовершенное. Прежде всего, одна из причин применения промежутков — избегание выделения памяти. Но обратите внимание, что наш метод Split создает экземпляр **List<Range>**, добавляет в него элементы и преобразует список в массив. Такие действия вовлекают, *по меньшей мере*, два выделения памяти, а также операцию копирования памяти.

Решение проблемы заключается в том, чтобы отказаться от списка и массива в пользу одностороннего перечислителя. С перечислителем труднее работать, но он может быть сделан без выделений памяти с использованием структур:

```

// Мы обязаны определить это как ref struct,
// потому что _input является ссылочной структурой.
public readonly ref struct CharSpanSplitter
{
    readonly ReadOnlySpan<char> _input;
    public CharSpanSplitter (ReadOnlySpan<char> input) => _input = input;
    public Enumerator GetEnumerator() => new Enumerator (_input);
    public ref struct Enumerator // Односторонний перечислитель
    {
        readonly ReadOnlySpan<char> _input;
        int _wordPos;
        public ReadOnlySpan<char> Current { get; private set; }
        public Rator (ReadOnlySpan<char> input)
        {
            _input = input;
            _wordPos = 0;
            Current = default;
        }
        public bool MoveNext()
        {
            for (int i = _wordPos; i <= _input.Length; i++)
                if (i == _input.Length || char.IsWhiteSpace (_input [i]))
                {
                    Current = _input [_wordPos..i];
                    _wordPos = i + 1;
                    return true;
                }
            return false;
        }
    }
}

public static class CharSpanExtensions
{
    public static CharSpanSplitter Split (this ReadOnlySpan<char> input)
        => new CharSpanSplitter (input);
    public static CharSpanSplitter Split (this Span<char> input)
        => new CharSpanSplitter (input);
}

```

Вот как его можно было бы применять:

```
var span = "the quick brown fox".AsSpan();
foreach (var word in span.Split())
{
    // word - экземпляр ReadOnlySpan<char>
}
```

За счет определения свойства `Current` и метода `MoveNext` наш перечисли́тель может работать с оператором `foreach` языка C# (см. раздел “Перечисление” в главе 4). Мы не обязаны реализовывать интерфейсы `IEnumerable<T>`/`IEnumerator<T>` (на самом деле это невозможно; ссылочные структуры не могут реализовывать интерфейсы). Мы жертвуем абстракцией ради микроопти-мизации.

Работа с выделяемой в стеке и неуправляемой памятью

Еще один эффективный прием микрооптимизации предусматривает сокращение нагрузки на сборщик мусора путем минимизации выделений памяти, основанных на куче. Это означает более широкое использование памяти в стеке — или даже неуправляемой памяти.

К сожалению, обычно такой подход требует переписывания кода с целью применения указателей. В случае нашего предыдущего примера, где суммировались элементы в массиве, пришлось бы написать другую версию, которая показана ниже:

```
unsafe int Sum (int* numbers, int length)
{
    int total = 0;
    for (int i = 0; i < length; i++) total += numbers [i];
    return total;
}
```

чтобы мы могли поступать следующим образом:

```
int* numbers = stackalloc int [1000]; // Выделить память под массив в стеке
int total = Sum (numbers, 1000);
```

Проблему решают промежутки: сконструировать экземпляр `Span<T>` или `ReadOnlySpan<T>` можно прямо из указателя:

```
int* numbers = stackalloc int [1000];
var span = new Span<int> (numbers, 1000);
```

Или за один шаг:

```
Span<int> numbers = stackalloc int [1000];
```

(Обратите внимание, что здесь не требуется `unsafe`.) Вспомним реализованный ранее метод `Sum`:

```
int Sum (ReadOnlySpan<int> numbers)
{
```

```
int total = 0;
int len = numbers.Length;
for (int i = 0; i < len; i++) total += numbers [i];
return total;
}
```

Данный метод в равной степени хорошо работает для промежутка, выделенного в стеке. Мы выиграли по трем пунктам:

- один и тот же метод функционирует с массивами и памятью, выделенной в стеке;
- память, выделенную в стеке, можно использовать с минимальным применением указателей;
- промежуток можно нарезать.



Компилятор достаточно интеллектуален, чтобы воспрепятствовать написанию метода, который выделяет память в стеке и возвращает ее вызывающему коду через экземпляр `Span<T>` или `ReadOnlySpan<T>`.

(Тем не менее, в других сценариях можно законно возвращать `Span<T>` или `ReadOnlySpan<T>`.)

Промежутки можно также использовать для создания оболочек, умещающих память, которая выделяется в неуправляемой куче. В приведенном ниже примере мы выделяем неуправляемую память с применением функции `Marshal.AllocHGlobal`, помещаем ее внутрь экземпляра `Span<char>` и копируем строку в неуправляемую память. В заключение мы задействуем структуру `CharSpanSplitter`, написанную в предыдущем разделе, для разбиения неуправляемой строки на слова:

```
var source = "The quick brown fox".AsSpan();
var ptr = Marshal.AllocHGlobal (source.Length * sizeof (char));
try
{
    var unmanaged = new Span<char> ((char*)ptr, source.Length);
    source.CopyTo (unmanaged);
    foreach (var word in unmanaged.Split())
        Console.WriteLine (word.ToString());
}
finally { Marshal.FreeHGlobal (ptr); }
```

Приятным бонусом является то, что индексатор структуры `Span<T>` выполняет проверку вхождения в границы, предотвращая переполнение буфера. Эта проверка применяется в случае корректного создания экземпляра `Span<T>`: в нашем примере такая защита будет утрачена, если промежуток получен неправильно:

```
var span = new Span<char> ((char*)ptr, source.Length * 2);
```

Кроме того, отсутствует защита от эквивалента висячего указателя, поэтому придется позаботиться о том, чтобы не обращаться к промежутку после освобождения его неуправляемой памяти с помощью функции `Marshal.FreeHGlobal`.



Способность к взаимодействию

В настоящей главе рассматриваются способы интеграции с низкоуровневыми (неуправляемыми) динамически подключаемыми библиотеками (Dynamic Link Library — DLL) и компонентами COM. Если не указано иное, то упоминаемые в главе типы находятся либо в пространстве имен System, либо в пространстве имен System.Runtime.InteropServices.

Обращение к низкоуровневым DLL-библиотекам

Технология *P/Invoke* (сокращение для Platform Invocation Services — службы вызова функций платформы) позволяет получать доступ к функциям, структурам и обратным вызовам в неуправляемых DLL-библиотеках (*совместно используемых библиотеках* в Unix).

Например, рассмотрим функцию `MessageBox`, которая определена в DLL-библиотеке Windows по имени `user32.dll` следующим образом:

```
int MessageBox (HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Эту функцию можно вызывать напрямую, объявив статический метод с тем же именем, применив ключевое слово `extern` и добавив атрибут `DllImport`:

```
using System;
using System.Runtime.InteropServices;

MessageBox (IntPtr.Zero,
           "Please do not press this again.", "Attention", 0);
           // Не нажмите это снова.

[DllImport("user32.dll")]
static extern int MessageBox (IntPtr hWnd, string text, string caption,
                           int type);
```

Классы `MessageBox` в пространствах имен `System.Windows` и `System.Windows.Forms` сами вызывают подобные неуправляемые методы.

Вот пример с `DllImport` для Ubuntu Linux:

```
Console.WriteLine ("User ID: {getuid()}");
[DllImport("libc")]
static extern uint getuid();
```

Среда CLR включает маршализатор, которому известно, как преобразовывать параметры и возвращаемые значения между типами .NET и неуправляемыми типами. В примере для Windows параметры `int` транслируются прямо в четырехбайтовые целые числа, которые ожидает функция, а строковые параметры преобразуются в массивы символов Unicode (в кодировке UTF-16), завершающиеся символом `null`. Структура `IntPtr` предназначена для инкапсуляции неуправляемого дескриптора и занимает 32 бита на 32-разрядных платформах и 64 бита на 64-разрядных plataформах. Похожая трансляция происходит и в Unix. (Начиная с версии C# 9, можно также использовать тип `nint`, который отображается на `IntPtr`.)

Маршализация типов и параметров

Маршализация общих типов

На неуправляемой стороне для представления необходимого типа данных может существовать более одного способа. Скажем, строка может содержать однобайтовые символы ANSI или символы Unicode в кодировке UTF-16 и предваряться в качестве префикса значением длины, завершаться символом `null` либо иметь фиксированную длину. С помощью атрибута `MarshalAs` маршализатору CLR сообщается используемый вариант, так что он обеспечит корректную трансляцию. Ниже показан пример:

```
[DllImport("...")]
static extern int Foo ( [MarshalAs(UnmanagedType.LPStr)] string s );
```

Перечисление `UnmanagedType` включает все типы Win32 и COM, распознаваемые маршализатором. В этом случае маршализатору указано на необходимость трансляции в тип `LPStr`, который является строкой, завершающейся `null`, с однобайтовыми символами ANSI.

На стороне .NET также имеется выбор относительно того, какой тип данных применять. Например, неуправляемые дескрипторы могут отображаться на тип `IntPtr`, `int`, `uint`, `long` или `ulong`.



Большинство неуправляемых дескрипторов инкапсулирует адрес или указатель и потому должно отображаться на `IntPtr` для совместимости с 32- и 64-разрядными операционными системами. Типичным примером может служить `HWND`.

Довольно часто функции Win32 и POSIX поддерживают целочисленный параметр, который принимает набор констант, определенных в заголовочном

файле C++, таком как `WinUser.h`. Вместо определения в виде простых констант C# их можно представить как члены перечисления. Использование перечисления может дать в результате более аккуратный код и увеличить статическую безопасность типов. В разделе “Совместно используемая память” далее в главе будет приведен пример.



При установке Microsoft Visual Studio удостоверьтесь, что устанавливаете такие заголовочные файлы C++ — даже если в категории C++ не выбрано ничего другого. Именно здесь определены все низкоуровневые константы Win32. Выяснить местонахождение всех заголовочных файлов можно, поискав файлы `*.h` в каталоге программ Visual Studio.

Стандарт POSIX в среде Unix определяет имена констант, но индивидуальные реализации совместимых с POSIX систем Unix могут присваивать этим константам отличающиеся числовые значения. Вы должны применять корректное числовое значение для выбранной операционной системы. Аналогичным образом POSIX определяет стандарт для структур, используемых в вызовах взаимодействия. Порядок следования полей в структуре стандартом не фиксируется и реализация Unix может добавлять дополнительные поля. Заголовочные файлы C++, определяющие функции и типы, часто устанавливаются в `/usr/include` или `/usr/local/include`.

Получение строк из неуправляемого кода обратно в .NET требует проведения некоторых действий по управлению памятью. Маршализатор выполняет такую работу автоматически, если внешний метод объявлен как принимающий объект `StringBuilder`, а не `string`:

```
StringBuilder s = new StringBuilder (256);
GetWindowsDirectory (s, 256);
Console.WriteLine (s);

[DllImport("kernel32.dll")]
static extern int GetWindowsDirectory (StringBuilder sb, int maxChars);
```

В среде Unix он работает похожим образом. В следующем коде вызывается `getcwd` для возвращения текущего каталога:

```
var sb = new StringBuilder (256);
Console.WriteLine (getcwd (sb, sb.Capacity));

[DllImport("libc")]
static extern string getcwd (StringBuilder buf, int size);
```

Несмотря на удобство использования типа `StringBuilder`, с ним связана некоторая неэффективность, т.к. среде CLR приходится выполнять дополнительные выделения памяти и копирование. В “горячих” точках, критичных к производительности, этих накладных расходов можно избежать за счет применения `char[]` вместо `StringBuilder`:

```
[DllImport ("kernel32.dll", CharSet = CharSet.Unicode)]
static extern int GetWindowsDirectory (char[] buffer, int maxChars);
```

Обратите внимание, что в атрибуте DllImport должен быть указан параметр CharSet. Кроме того, после вызова функции выходную строку потребуется усечь до нужной длины. Добиться указанной цели, одновременно сведя к минимуму выделения памяти, можно с использованием пула массивов (см. раздел “Организация пула массивов” в главе 12):

```
string GetWindowsDirectory()
{
    var array = ArrayPool<char>.Shared.Rent(256);
    try
    {
        int length = GetWindowsDirectory(array, 256);
        return new string(array, 0, length).ToString();
    }
    finally { ArrayPool<char>.Shared.Return(array); }
}
```

(Разумеется, приведенный пример надуман, поскольку каталог Windows можно получить с помощью встроенного метода Environment.GetFolderPath.)



Если вы не уверены, каким образом должен вызываться отдельный метод Win32 или Unix, тогда поищите пример его вызова в Интернете, указав в качестве строки поиска имя метода и слово DllImport. На сайте <http://www.pinvoke.net> стараются документировать все сигнатуры Win32.

Маршализация классов и структур

Иногда неуправляемому методу необходимо передавать структуру. Например, метод GetSystemTime в API-интерфейсе Win32 определен следующим образом:

```
void GetSystemTime (LPSYSTEMTIME lpSystemTime);
```

Тип LPSYSTEMTIME соответствует такой структуре C:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Чтобы вызвать метод GetSystemTime, мы должны определить класс или структуру .NET для соответствия показанной выше структуре C:

```
using System;
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential)]
```

```

class SystemTime
{
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milliseconds;
}

```

Атрибут `StructLayout` указывает маршализатору, как следует отображать каждое поле на его неуправляемый эквивалент. Член перечисления `LayoutKind.Sequential` означает, что поля должны выравниваться последовательно по границам размеров пакета (объясняется ниже), точно так же как это было бы в структуре С. Имена полей роли не играют; важен только порядок следования полей.

Теперь метод `GetSystemTime` можно вызывать:

```

SystemTime t = new SystemTime();
GetSystemTime (t);
Console.WriteLine (t.Year);

[DllImport("kernel32.dll")]
static extern void GetSystemTime (SystemTime t);

```

А вот как его вызывать в среде Unix:

```

Console.WriteLine (GetSystemTime());
static DateTime GetSystemTime()
{
    DateTime startOfUnixTime =
        new DateTime(1970, 1, 1, 0, 0, 0, System DateTimeKind.Utc);
    Timespec tp = new Timespec();
    int success = clock_gettime (0, ref tp);
    if (success != 0) throw new Exception ("Error checking the time.");
                                                // Ошибка при проверке времени
    return startOfUnixTime.AddSeconds (tp.tv_sec).ToLocalTime();
}

[DllImport("libc")]
static extern int clock_gettime (int clk_id, ref Timespec tp);

[StructLayout(LayoutKind.Sequential)]
struct Timespec
{
    public long tv_sec;    /* секунды */
    public long tv_nsec;   /* наносекунды */
}

```

В языках С и C# поля в объекте располагаются со смещением в *n* байтов, начиная с адреса объекта. Разница в том, что в программе C# среда CLR находит такое смещение с применением маркера поля, а в случае С имена полей компилируются прямо в смещения. Например, в языке С поле `wDay` — просто маркер для представления чего-либо, находящегося по адресу экземпляра `SystemTime` плюс 24 байта.

Для ускорения доступа каждое поле размещается со смещением, кратным размеру поля. Однако используемый множитель ограничен максимумом в x байтов, где x представляет собой *размер пакета*. В текущей реализации стандартный размер пакета составляет 8 байтов, так что структура, содержащая поле `sbyte`, за которым следует (8-байтовое) поле `long`, занимает 16 байтов, и 7 байтов, следующих за `sbyte`, расходуются впустую. Потери подобного рода можно снизить или вообще устраниТЬ, указывая размер пакета через свойство `Pack` в атрибуте `StructLayout`: это обеспечивает выравнивание по смещениям, кратным заданному размеру пакета. Таким образом, при размере пакета, равном 1, только что описанная структура будет занимать только 9 байтов. В качестве размера пакета можно указывать 1, 2, 4, 8 или 16 байтов.

Атрибут `StructLayout` также позволяет задавать явные смещения полей (как показано в разделе “Эмуляция объединения С” далее в главе).

Маршализация параметров `in` и `out`

В предыдущем примере мы реализовали `SystemTime` в виде класса. Вместо него можно было бы выбрать структуру при условии, что метод `GetSystemTime` объявлен с параметром `ref` или `out`:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime (out SystemTime t);
```

В большинстве случаев семантика направленных параметров C# работает одинаково и с внешними методами. Параметры, передаваемые по значению, копируют параметры `in`, параметры `ref` в C# копируют параметры `in/out`, а параметры `out` в C# копируют параметры `out`. Тем не менее, существует ряд исключений для типов, которые имеют специальные преобразования. Например, классы массивов и класс `StringBuilder` требуют копирования при выдаче из функции, поэтому они являются `in/out`. Иногда такое поведение удобно переопределять посредством атрибутов `In` и `Out`. Скажем, если массив должен допускать только чтение, то атрибут `In` указывает, что в функцию поступает только копия массива, но выводиться он из функции не будет:

```
static extern void Foo ( [In] int[] array);
```

Соглашения о вызовах

Неуправляемые методы принимают аргументы и возвращают значения через стек и (необязательно) через регистры ЦП. Поскольку для этого существует несколько способов, появились различные протоколы, которые известны как соглашения о вызовах. В текущий момент среда CLR поддерживает три соглашения о вызовах: `StdCall`, `Cdecl` и `ThisCall`.

По умолчанию среда CLR использует стандартное соглашение о вызове, принятое для платформы. В Windows им является `StdCall`, а в Linux x86 — `Cdecl`.

Если неуправляемый метод не соответствует такому стандартному соглашению, тогда можно явно указать его соглашение о вызове, как показано ниже:

```
[DllImport ("MyLib.dll", CallingConvention=CallingConvention.Cdecl)]
static extern void SomeFunc (...)
```

Несколько запутанно именованный член `CallingConvention.WinApi` обозначает стандартное соглашение о вызове, принятое для платформы.

Обратные вызовы из неуправляемого кода

Язык C# также разрешает внешним функциям обращаться к коду C# через обратные вызовы. Есть два способа выполнения обратных вызовов:

- через указатели на функции (начиная с версии C# 9);
- посредством делегатов.

В целях иллюстрации мы будем вызывать определенную в библиотеке `User32.dll` следующую функцию Windows, которая предназначена для перечисления всех высокогорневых оконных дескрипторов:

```
BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

`WNDENUMPROC` — это обратный вызов, который последовательно запускается с дескриптором каждого окна (или до тех пор, пока обратный вызов не возвратит `false`). Вот его определение:

```
BOOL CALLBACK EnumWindowsProc (HWND hWnd, LPARAM lParam);
```

Обратные вызовы с помощью указателей на функции

Начиная с версии C# 9, самый простой и наиболее эффективный способ в ситуации, когда обратный вызов является статическим методом, предусматривает применение *указателя на функцию*. В случае обратного вызова `WNDENUMPROC` мы можем использовать указатель на функцию такого вида:

```
delegate*<IntPtr, IntPtr, bool>
```

Он обозначает функцию, которая принимает два аргумента типа `IntPtr` и возвращает значение `bool`. Затем с помощью операции & нашей функции можно передать статический метод:

```
using System;
using System.Runtime.InteropServices;
unsafe
{
    EnumWindows (&PrintWindow, IntPtr.Zero);
    [DllImport ("user32.dll")]
    static extern int EnumWindows (
        delegate*<IntPtr, IntPtr, bool> hWnd, IntPtr lParam);

    static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
    {
        Console.WriteLine (hWnd.ToInt64 ());
        return true;
    }
}
```

Указатели на функции требуют, чтобы обратный вызов был статическим методом (или статической функцией, как в данном примере).

UnmanagedCallersOnly

Применив ключевое слово `unmanaged` к указателю на функцию и атрибут `[UnmanagedCallersOnly]` к методу обратного вызова, можно повысить производительность:

```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

unsafe
{
    EnumWindows (&PrintWindow, IntPtr.Zero);
    [DllImport ("user32.dll")]
    static extern int EnumWindows (
        delegate* unmanaged <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);
    [UnmanagedCallersOnly]
    static byte PrintWindow (IntPtr hWnd, IntPtr lParam)
    {
        Console.WriteLine (hWnd.ToInt64 ());
        return 1;
    }
}
```

Атрибут `[UnmanagedCallersOnly]` отмечает метод `PrintWindow` как такой, который может быть вызван *только* из неуправляемого кода, позволяя исполняющей среде двигаться кратчайшими путями. Обратите внимание, что мы также изменили возвращаемый тип метода с `bool` на `byte`: причина в том, что методы, к которым применяется атрибут `[UnmanagedCallersOnly]`, могут использовать в своих сигнтурах только *преобразуемые* (*blittable*) типы значений. Преобразуемые типы — это типы, которые не требуют какой-то специальной логики маршализации, потому что они представлены идентично в управляемом и неуправляемом мирах. К ним относятся примитивные целочисленные типы, `float`, `double` и структуры, содержащие только преобразуемые типы. Тип `char` — тоже преобразуемый, если он является частью структуры с атрибутом `StructLayout`, указывающим `CharSet.Unicode`:

```
[StructLayout (LayoutKind.Sequential, CharSet=CharSet.Unicode)]
```

Нестандартные соглашения о вызовах

По умолчанию компилятор предполагает, что неуправляемый обратный вызов следует стандартному соглашению о вызове, принятому для платформы. Если это не так, тогда можно явно указать его соглашение о вызове через параметр `CallConvs` атрибута `[UnmanagedCallersOnly]`:

```
[UnmanagedCallersOnly (CallConvs = new[] { typeof (CallConvStdcall) })]
static byte PrintWindow (IntPtr hWnd, IntPtr lParam) ...
```

Также потребуется обновить тип указателя на функцию, вставив после ключевого слова `unmanaged` специальный модификатор:

```
delegate* unmanaged[Stdcall] <IntPtr, IntPtr, byte> hWnd, IntPtr lParam);
```



Компилятор позволяет помещать любой идентификатор (вроде XYZ) внутрь квадратных скобок при условии, что имеется тип .NET по имени CallConvXYZ (он воспринимается исполняющей средой и совпадает с тем, который был указан во время применения атрибута [UnmanagedCallersOnly]). Это облегчает добавление в будущем новых соглашений о вызовах разработчиками из Microsoft.

В данном случае был указан вариант StdCall, принятый по умолчанию для платформы Windows (для платформы Linux x86 по умолчанию принят вариант Cdecl). Ниже перечислены все варианты, которые поддерживаются в текущий момент:

Имя	Модификатор <code>unmanaged</code>	Поддерживающий тип
Stdcall	<code>unmanaged[Stdcall]</code>	CallConvStdcall
Cdecl	<code>unmanaged[Cdecl]</code>	CallConvCdecl
ThisCall	<code>unmanaged[Thiscall]</code>	CallConvThiscall

Обратные вызовы с помощью делегатов

Неуправляемые обратные вызовы могут выполняться также с помощью делегатов. Такой подход работает во всех версиях C# и позволяет обратным вызовам ссылаться на методы экземпляра.

Для начала мы объявим тип делегата с сигнатурой, которая совпадает с обратным вызовом. Затем можно передать экземпляр этого делегата внешнему методу:

```
class CallbackFun
{
    delegate bool EnumWindowsCallback (IntPtr hWnd, IntPtr lParam);
    [DllImport("user32.dll")]
    static extern int EnumWindows (EnumWindowsCallback hWnd, IntPtr lParam);
    static bool PrintWindow (IntPtr hWnd, IntPtr lParam)
    {
        Console.WriteLine (hWnd.ToInt64 ());
        return true;
    }
    static readonly EnumWindowsCallback printWindowFunc = PrintWindow;
    static void Main() => EnumWindows (printWindowFunc, IntPtr.Zero);
}
```

По иронии судьбы использовать делегаты для неуправляемых обратных вызовов небезопасно, поскольку легко угодить в ловушку, позволив обратному вызову произойти после того, как экземпляр делегата покинул область видимости (в этот момент делегат становится пригодным для сборки мусора). Результатом может оказаться наихудшее исключение времени выполнения — то, которое не имеет полезной трассировки стека. В случае обратных вызовов в виде статических методов описанной ситуации можно избежать, присваивая экземпляр делегата статическому полю, допускающему только чтение (как делалось в при-

мере выше). Обратным вызовам в виде методов экземпляра шаблон такого рода не поможет, поэтому нужно обеспечить, чтобы на протяжении любого потенциального обратного вызова поддерживалась хотя бы одна ссылка на экземпляр делегата. Но даже тогда при возникновении ошибки в неуправляемой функции, из-за которой она инициирует обратный вызов после того, как ей было запрещено это делать, возможно, придется иметь дело с исключением без трассировки стека. Решение проблемы предусматривает определение уникального типа делегата для каждой неуправляемой функции: прием полезен с точки зрения диагностики, т.к. в исключении будет сообщаться тип делегата.

Чтобы изменить стандартное соглашение о вызове, принятое для платформы, к делегату можно применить атрибут [UnmanagedFunctionPointer]:

```
[UnmanagedFunctionPointer (CallingConvention.Cdecl)]
delegate void MyCallback (int foo, short bar);
```

Эмуляция объединения C

Каждое поле в структуре получает достаточно места для хранения своих данных. Рассмотрим структуру, содержащую одно поле типа int и одно поле типа char. Поле int, по всей видимости, начнется со смещения 0 и гарантированно займет, по меньшей мере, четыре байта. Таким образом, поле char начнется со смещения минимум 4. Если по какой-то причине поле char начнется со смещения 2, то присваивание значения полю char приведет к изменению значения поля int. Похоже на хаос, не так ли? Как ни странно, в языке C поддерживается разновидность структуры под названием *объединение*, которая делает именно то, что было описано. Эмулировать объединение в языке C# можно с использованием значения LayoutKind.Explicit и атрибута FieldOffset.

Придумать сценарий, когда объединение может оказаться полезным, может быть непросто. Тем не менее, представим, что необходимо воспроизвести ноту на внешнем синтезаторе. API-интерфейс Windows Multimedia предоставляет функцию, которая делает это через протокол MIDI:

```
[DllImport ("winmm.dll")]
public static extern uint midiOutShortMsg (IntPtr handle, uint message);
```

Второй аргумент, message, описывает, какую ноту необходимо воспроизвести. Проблема связана с конструкцией этого 32-битного целого числа без знака: внутренне оно разделено на байты, представляющие канал MIDI, ноту и скорость звучания. Одно из решений предусматривает сдвиг и применение масок через побитовые операции <<, >>, & и | для преобразования таких байтов в и из 32-битного “упакованного” сообщения. Однако намного проще определить структуру с явной компоновкой:

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
    [FieldOffset(0)] public uint PackedMsg; // Длина 4 байта
    [FieldOffset(0)] public byte Channel; // FieldOffset также 0
    [FieldOffset(1)] public byte Note;
    [FieldOffset(2)] public byte Velocity;
}
```

Поля `Channel`, `Note` и `Velocity` преднамеренно пересекаются с 32-битным упакованным сообщением, что позволяет осуществлять чтение и запись, используя либо то, либо другое. Для поддержания полей в синхронизированном состоянии никаких дополнительных вычислений не потребуется:

```
NoteMessage n = new NoteMessage();
Console.WriteLine (n.PackedMsg); // 0
n.Channel = 10;
n.Note = 100;
n.Velocity = 50;
Console.WriteLine (n.PackedMsg); // 3302410
n.PackedMsg = 3328010;
Console.WriteLine (n.Note); // 200
```

Совместно используемая память

Размещенные в памяти файлы, или *совместно используемая память* — это функциональная возможность Windows, которая позволяет множеству процессов на одном компьютере совместно использовать данные. Совместно используемая память является исключительно быстрой и в отличие от каналов предлагает произвольный доступ к общим данным. В главе 15 было показано, как применять класс `MemoryMappedFile` для доступа к размещенным в памяти файлам; вызов методов Win32 напрямую будет хорошим способом демонстрации уровня P/Invoke.

Функция `CreateFileMapping` в API-интерфейсе Win32 выделяет совместно используемую память. Ей необходимо указать, сколько байтов требуется, а также имя, под которым будет идентифицироваться совместно используемая память. Затем другое приложение может подписаться на данную память, вызвав функцию `OpenFileMapping` с этим именем. Обе функции возвращают *дескриптор*, который можно преобразовать в указатель с помощью вызова функции `MapViewOfFile`.

Ниже представлен класс, инкапсулирующий доступ к совместно используемой памяти:

```
using System;
using System.Runtime.InteropServices;
using System.ComponentModel;
public sealed class SharedMem : IDisposable
{
    // Здесь мы используем перечисления, потому что они безопаснее констант
    enum FileProtection : uint           // константы из winnt.h
    {
        ReadOnly = 2,
        ReadWrite = 4
    }
    enum FileRights : uint               // константы из WinBASE.h
    {
        Read = 4,
        Write = 2,
        ReadWrite = Read + Write
    }
}
```

```

static readonly IntPtr NoFileHandle = new IntPtr (-1);

[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr CreateFileMapping (IntPtr hFile,
                                       int lpAttributes,
                                       FileProtection flProtect,
                                       uint dwMaximumSizeHigh,
                                       uint dwMaximumSizeLow,
                                       string lpName);

[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr OpenFileMapping (FileRights dwDesiredAccess,
                                      bool bInheritHandle,
                                      string lpName);

[DllImport ("kernel32.dll", SetLastError = true)]
static extern IntPtr MapViewOfFile (IntPtr hFileMappingObject,
                                    FileRights dwDesiredAccess,
                                    uint dwFileOffsetHigh,
                                    uint dwFileOffsetLow,
                                    uint dwNumberOfBytesToMap);

[DllImport ("Kernel32.dll", SetLastError = true)]
static extern bool UnmapViewOfFile (IntPtr map);

[DllImport ("kernel32.dll", SetLastError = true)]
static extern int CloseHandle (IntPtr hObject);

IntPtr fileHandle, fileMap;

public IntPtr Root => fileMap;

public SharedMem (string name, bool existing, uint sizeInBytes)
{
    if (existing)
        fileHandle = OpenFileMapping (FileRights.ReadWrite, false, name);
    else
        fileHandle = CreateFileMapping (NoFileHandle, 0,
                                       FileProtection.ReadWrite,
                                       0, sizeInBytes, name);

    if (fileHandle == IntPtr.Zero)
        throw new Win32Exception();

    // Получить отображение с возможностью чтения/записи для всего файла
    fileMap = MapViewOfFile (fileHandle, FileRights.ReadWrite, 0, 0, 0);

    if (fileMap == IntPtr.Zero)
        throw new Win32Exception();
}

public void Dispose()
{
    if (fileMap != IntPtr.Zero) UnmapViewOfFile (fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle (fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
}
}

```

В приведенном примере мы указываем `SetLastError = true` в методах `DllImport`, которые используют протокол `SetLastError` для выдачи кодов ошибок. Это обеспечит заполнение исключения `Win32Exception` де-

тальными сведениями об ошибке, когда оно будет сгенерировано. (Вдобавок также появляется возможность запрашивать ошибку явно вызовом метода `Marshal.GetLastWin32Error`.)

Для демонстрации работы класса `SharedMem` понадобится запустить два приложения. Первое из них создает совместно используемую память следующим образом:

```
using (SharedMem sm = new SharedMem ("MyShare", false, 1000))
{
    IntPtr root = sm.Root;
    // Появился доступ к совместно используемой памяти
    Console.ReadLine(); // В этот момент мы запускаем второе приложение...
}
```

Второе приложение подписывается на совместно используемую память, конструируя объект `SharedMem` с тем же самым именем и передавая значение `true` в качестве аргумента `existing`:

```
using (SharedMem sm = new SharedMem ("MyShare", true, 1000))
{
    IntPtr root = sm.Root;
    // Появился доступ к той же самой совместно используемой памяти
    // ...
}
```

В конечном итоге каждая программа имеет объект `IntPtr` — указатель на одну и ту же неуправляемую память. Теперь два приложения должны каким-то образом выполнять чтение и запись в память через имеющийся общий указатель. Один из подходов предполагает построение сериализируемого класса, который инкапсулирует все совместно используемые данные, после чего сериализирует (и десериализирует) данные в неуправляемую память с применением класса `UnmanagedMemoryStream`. Однако при наличии большого объема данных такой прием неэффективен. Представьте себе ситуацию, когда класс совместно используемой памяти имеет мегабайт данных, но нужно обновить только одно целочисленное значение. Более удачный подход предусматривает определение конструкции совместно используемых данных в виде структуры, и затем ее отображение прямо на совместно используемую память. Мы обсудим это в следующем разделе.

Отображение структуры на неуправляемую память

Структура, для которой в атрибуте `StructLayout` указано значение `Sequential` или `Explicit`, может отображаться прямо на неуправляемую память. Рассмотрим показанную ниже структуру:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
    public int Value;
    public char Letter;
    public fixed float Numbers [50];
}
```

Директива `fixed` позволяет определять массивы типов значений фиксированной длины встроенным образом, что как раз и создает область `unsafe`. Пространство в данной структуре выделяется встроенным образом для 50 чисел с плавающей точкой. В отличие от стандартных массивов C# член `Numbers` — не ссылка на массив, а сам массив. Если выполнить следующий код:

```
static unsafe void Main() => Console.WriteLine (sizeof (MySharedData));
```

то на консоль выводится результат 208: 50 четырехбайтовых значений `float` плюс четыре байта для поля `Value` типа `int` плюс два байта для поля `Letter` типа `char`. Общее количество байтов, равное 206, округляется до 208 из-за того, что значения `float` выравниваются по четырехбайтовым границам (четыре байта — размер типа `float`).

Проще всего продемонстрировать использование структуры `MySharedData` в контексте `unsafe` на примере памяти, выделенной в стеке:

```
MySharedData d;  
MySharedData* data = &d; // Получить адрес d  
  
data->Value = 123;  
data->Letter = 'X';  
data->Numbers[10] = 1.45f;
```

или:

```
// Распределить массив в стеке:  
MySharedData* data = stackalloc MySharedData[1];  
data->Value = 123;  
data->Letter = 'X';  
data->Numbers[10] = 1.45f;
```

Разумеется, мы здесь не демонстрируем ничего такого, чего нельзя было бы достичь в управляемом контексте. Но предположим, что мы хотим хранить экземпляр `MySharedData` в *неуправляемой куче*, т.е. за пределами действия сборщика мусора CLR. Именно в таких случаях указатели становятся по-настоящему полезными:

```
MySharedData* data = (MySharedData*)  
Marshal.AllocHGlobal (sizeof (MySharedData)).ToPointer();  
data->Value = 123;  
data->Letter = 'X';  
data->Numbers[10] = 1.45f;
```

Метод `Marshal.AllocHGlobal` выделяет память в *неуправляемой куче*. Вот как позже освободить ту же самую память:

```
Marshal.FreeHGlobal (new IntPtr (data));
```

(Если забыть об освобождении этой памяти, тогда в результате возникнет хорошо известная утечка памяти.)



Начиная с версии .NET 6, для выделения и освобождения *неуправляемой памяти* можно применять класс `NativeMemory`, который использует более новый (и лучший) базовый API-интерфейс, чем `AllocHGlobal`, а также включает методы для выполнения выровненных выделений памяти.

В соответствии с ее именем мы будем применять структуру MySharedData вместе с классом SharedMem, написанным в предыдущем разделе. В следующей программе выделяется блок совместно используемой памяти, на который затем отображается структура MySharedData:

```
static unsafe void Main()
{
    using (SharedMem sm = new SharedMem ("MyShare", false,
                                         (uint) sizeof (MySharedData)))
    {
        void* root = sm.Root.ToPointer();
        MySharedData* data = (MySharedData*) root;
        data->Value = 123;
        data->Letter = 'X';
        data->Numbers[10] = 1.45f;
        Console.WriteLine ("Written to shared memory");
        // Записано в совместно используемую память
        Console.ReadLine();
        Console.WriteLine ("Value is " + data->Value);           // Поле Value
        Console.WriteLine ("Letter is " + data->Letter);         // Поле Letter
        Console.WriteLine ("11th Number is " + data->Numbers[10]);
        // 11-й элемент Numbers
        Console.ReadLine();
    }
}
```



Вместо SharedMem можно применять встроенный класс MemoryMappedFile:

```
using (MemoryMappedFile mmFile =
       MemoryMappedFile.CreateNew ("MyShare", 1000))
using (MemoryMappedViewAccessor accessor =
       mmFile.CreateViewAccessor())
{
    byte* pointer = null;
    accessor.SafeMemoryMappedViewHandle.AcquirePointer
        (ref pointer);
    void* root = pointer;
    ...
}
```

Ниже представлена вторая программа, которая присоединяется к той же самой совместно используемой памяти и читает значения, записанные первой программой (она должна быть запущена, пока первая программа ожидает в операторе ReadLine, т.к. после выхода из оператора using объект совместно используемой памяти освобождается):

```
static unsafe void Main()
{
    using (SharedMem sm = new SharedMem ("MyShare", true,
                                         (uint) sizeof (MySharedData)))
    {
        void* root = sm.Root.ToPointer();
        MySharedData* data = (MySharedData*) root;
```

```

Console.WriteLine ("Value is " + data->Value);           // Поле Value
Console.WriteLine ("Letter is " + data->Letter);         // Поле Letter
Console.WriteLine ("11th Number is " + data->Numbers[10]);
               // 11-й элемент Numbers

// Нама очередь обновлять значения в совместно используемой памяти
data->Value++;
data->Letter = '!';
data->Numbers[10] = 987.5f;
Console.WriteLine ("Updated shared memory");
               // Обновлено в совместно используемой памяти
Console.ReadLine();
}
}

```

Далее приведен вывод из обеих программ.

Первая программа:

```

Written to shared memory
Value is 124
Letter is !
11th Number is 987.5

```

Вторая программа:

```

Value is 123
Letter is X
11th Number is 1.45
Updated shared memory

```

Не стоит пугаться указателей: программисты на языке C++ применяют указатели в приложениях повсеместно и способны заставить их работать в любой ситуации. Во всяком случае, большую часть времени. Такой вид использования является сравнительно простым.

Наш пример небезопасен по другой причине. Мы не принимали во внимание проблемы безопасности в отношении потоков (или, выражаясь точнее — безопасности в отношении процессов), которые возникают в ситуации, когда две программы получают доступ к одной и той же памяти одновременно. Чтобы задействовать такой прием в производственном приложении, к полям Value и Letter структуры MySharedData потребуется добавить ключевое слово volatile, чтобы предотвратить кэширование этих полей компилятором ЛТ (или оборудованием в регистрах центрального процессора). Вдобавок по мере выхода взаимодействия с полями за рамки тривиального почти наверняка придется защищать доступ к ним с помощью межпроцессного объекта Mutex — точно так же, как мы бы применяли операторы lock для защиты доступа к полям в многопоточной программе. Безопасность к потокам подробно обсуждалась в главе 21.

fixed и fixed { . . . }

Одно из ограничений отображения структур напрямую в память связано с тем, что структуры могут содержать только неуправляемые типы. Если необходимо совместно использовать, например, строковые данные, тогда должен при-

меняться фиксированный массив символов, что означает ручное преобразование в тип `string` и из него. Вот как это делать:

```
[StructLayout (LayoutKind.Sequential)]
unsafe struct MySharedData
{
    ...
    // Выделить пространство для 200 символов (т.е. 400 байтов).
    const int MessageSize = 200;
    fixed char message [MessageSize];

    // Вероятно, данный код имеет смысл поместить во вспомогательный класс:
    public string Message
    {
        get { fixed (char* cp = message) return new string (cp); }
        set
        {
            fixed (char* cp = message)
            {
                int i = 0;
                for (; i < value.Length && i < MessageSize - 1; i++)
                    cp [i] = value [i];
                // Добавить завершающий символ null.
                cp [i] = '\0';
            }
        }
    }
}
```



Понятие вроде ссылки на фиксированный массив отсутствует; взамен вы получаете указатель. При индексации в фиксированном массиве вы на самом деле выполняете арифметические действия над указателями.

С помощью первого случая использования ключевого слова `fixed` мы выделяем пространство для 200 символов встроенным в структуру образом. Когда ключевое слово `fixed` позже применяется в определении свойства, оно имеет другой смысл (что иногда запутывает). В такой ситуации `fixed` инструктирует среду CLR о необходимости закрепления объекта, так что если принимается решение о проведении сборки мусора внутри блока `fixed`, то внутренняя структура не должна перемещаться в рамках кучи (поскольку по ее содержимому будет производиться итерация через прямые указатели в памяти). Глядя на приведенную выше программу, может возникнуть вопрос о том, как вообще структура `MySharedData` может переместиться в памяти, если она располагается не в куче, а в неуправляемой памяти, к которой сборщик мусора не имеет никакого отношения? Однако компилятору ничего не известно о данном факте, и он предполагает, что вы *можете* использовать `MySharedData` в управляемом контексте, поэтому настоятельно требует добавления ключевого слова `fixed`, чтобы сделать код `unsafe` безопасным в управляемых контекстах. И компилятор полностью прав — взгляните, насколько легко поместить структуру `MySharedData` в кучу:

```
object obj = new MySharedData();
```

Результатом будет упакованная структура `MySharedData`, которая находится в куче и может быть перемещена во время сборки мусора.

Рассмотренный пример проиллюстрировал, как строка может быть представлена в структуре, отображаемой на неуправляемую память. Для более сложных типов также доступен вариант применения существующего кода сериализации. Единственное условие — сериализированные данные никогда не должны превышать по длине выделенное для них пространство в структуре, иначе это приведет к непреднамеренному объединению с последующими полями.

Взаимодействие с COM

Исполняющая среда .NET предлагает специальную поддержку COM, разрешая работать с объектами COM в .NET и наоборот. Поддержка COM доступна только в Windows.

Назначение COM

Модель компонентных объектов (Component Object Model — COM) представляет собой двоичный стандарт для взаимодействия с библиотеками, который был выпущен Microsoft в 1993 году. Мотивацией к созданию COM была необходимость предоставления компонентам возможности взаимодействия друг с другом в независимой от языка и безразличной к версиям манере. До появления COM подход, применяемый в Windows, заключался в опубликовании DLL-библиотек, которые объявляли структуры и функции с использованием языка программирования С. Такой подход был не только специфичным к языку, но и достаточно хрупким. Спецификация типа в библиотеке подобного рода неотделима от его реализации: даже добавление к структуре нового поля разрушало ее спецификацию.

Элегантность COM заключалась в отделении спецификации типа от его реализации через конструкцию, известную как *интерфейс COM*. Технология COM также позволила вызывать методы на *объектах*, поддерживающих состояние — не ограничиваясь простыми вызовами процедур.



В определенном смысле модель программирования для .NET является эволюцией принципов программирования для COM: платформа .NET также упрощает разработку на многочисленных языках и позволяет двоичным компонентам развиваться, не нарушая работу приложений, которые от них зависят.

Основы системы типов COM

Система типов COM вращается вокруг интерфейсов. Интерфейс COM довольно похож на интерфейс .NET, но получил большее распространение из-за того, что тип COM открывает свою функциональность *только* через интерфейс. Например, вот как мы могли бы объявить тип в мире .NET:

```
public class Foo
{
    public string Test() => "Hello, world";
}
```

Потребители типа Foo могут применять метод Test напрямую. И если позже будет изменена реализация метода Test, то повторная компиляция вызывающих сборок не потребуется. В таком отношении платформа .NET отделяет интерфейс от реализации, не делая интерфейсы обязательными. Можно было бы даже добавить перегруженную версию метода Test без нарушения работы вызывающих компонентов:

```
public string Test (string s) => $"Hello, world {s}";
```

В мире COM для достижения такого же уровня развязки класс Foo открывает свою функциональность через интерфейс. Таким образом, в библиотеке типов Foo будет присутствовать интерфейс, подобный представленному ниже:

```
public interface IFoo { string Test(); }
```

(Мы иллюстрировали сказанное, показав интерфейс C# — не интерфейс COM. Тем не менее, принцип остается тем же самым, хотя связующий код отличается.)

Вызывающие компоненты будут затем взаимодействовать с IFoo, а не с Foo.

Когда дело доходит до добавления перегруженной версии метода Test, ситуация с технологией COM оказывается более сложной, чем с .NET. Во-первых, мы хотели бы избежать модификации интерфейса IFoo, т.к. это нарушило бы двоичную совместимость с предыдущей версией (один из принципов COM заключается в том, что интерфейсы после опубликования являются *неизменяемыми*). Во-вторых, технология COM не поддерживает перегрузку методов. Решение состоит в том, чтобы обеспечить реализацию классом Foo *другого интерфейса*:

```
public interface IFoo2 { string Test (string s); }
```

(И снова для придания знакомого вида мы представили его в виде интерфейса .NET.)

Поддержка множества интерфейсов играет ключевую роль в возможности создания версий библиотек COM.

IUnknown и IDispatch

Все интерфейсы COM идентифицируются с помощью глобально уникального идентификатора (Globally Unique Identifier — GUID).

Корневым интерфейсом в COM является IUnknown; его обязаны реализовывать все объекты COM. Он имеет три метода:

- AddRef
- Release
- QueryInterface

Методы AddRef и Release предназначены для управления временем жизни, поскольку в COM используется подсчет ссылок, а не автоматическая сборка мусора (технология COM была спроектирована для работы с неуправляемыми объектами).

мым кодом, в котором автоматическая сборка мусора неосуществима). Метод `QueryInterface` возвращает ссылку на объект, который поддерживает данный интерфейс, если он способен делать это.

Чтобы стало возможным динамическое программирование (например, написание сценариев и автоматизация), объект COM может также реализовывать интерфейс `IDispatch`. В результате у динамических языков появляется возможность обращаться к объектам COM с применением позднего связывания — почти как с помощью `dynamic` в C# (хотя только для простых вызовов).

Обращение к компоненту COM из C#

Наличие встроенной в CLR поддержки для COM означает, что работать напрямую с интерфейсами `IUnknown` и `IDispatch` не придется. Взамен вы имеете дело с объектами CLR, а исполняющая среда маршализирует обращения к миру COM через *вызываемые оболочки времени выполнения* (Runtime-Callable Wrapper — RCW). Исполняющая среда также отвечает за управление временем жизни, вызывая методы `AddRef` и `Release` (когда объект .NET финализируется), и заботится о преобразованиях примитивных типов между двумя мирами. Преобразование типов гарантирует, что каждая сторона видит, например, целочисленные и строковые типы в знакомых ей формах.

Кроме того, необходим какой-нибудь способ доступа к оболочкам RCW в статически типизированной манере. Такая работа выполняется *типами взаимодействия с COM*. Типы взаимодействия с COM — это автоматически генерированные типы-посредники, которые открывают доступ к члену .NET для каждого члена COM. Инструмент импорта библиотек типов (`tlbimp.exe`) генерирует типы взаимодействия с COM в командной строке на основе выбранной библиотеки COM и компилирует их в *сборку взаимодействия с COM*.



Если компонент COM реализует сразу несколько интерфейсов, тогда инструмент `tlbimp.exe` генерирует одиночный тип, который содержит объединение членов из всех интерфейсов.

Сборку взаимодействия с COM можно создать в Visual Studio, открыв диалоговое окно `Add Reference` (Добавить ссылку) и выбрав нужную библиотеку на вкладке COM. Например, при наличии установленной программы Microsoft Excel добавление ссылки на библиотеку Microsoft Excel Object Library позволяет взаимодействовать с классами COM для Excel. Ниже приведен код, который создает и отображает рабочую книгу, после чего заполняет в ней ячейку:

```
using System;
using Excel = Microsoft.Office.Interop.Excel;

var excel = new Excel.Application();
excel.Visible = true;
excel.WindowState = Excel.XlWindowState.xlMaximized;
Excel.Workbook workBook = excel.Workbooks.Add();
((Excel.Range)excel.Cells[1, 1]).Font.FontStyle = "Bold";
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello World";
workBook.SaveAs(@"d:\temp.xlsx");
```



В настоящее время типы взаимодействия необходимо внедрять в разрабатываемое приложение (иначе исполняющая среда не найдет их во время выполнения). Щелкните на ссылке COM в проводнике решения Visual Studio и в окне свойств проекта установите параметр Embed Interop Types (Внедрять типы взаимодействия) в true или откройте файл .csproj и добавьте в него следующую строку (выделенную полужирным):

```
<ItemGroup>
    <COMReference Include="Microsoft.Office.Excel.dll">
        ...
        <EmbedInteropTypes>true</EmbedInteropTypes>
    </COMReference>
</ItemGroup>
```

Класс Excel.Application — это тип взаимодействия с COM, чьим типом времени выполнения является RCW. Когда мы обращаемся к свойствам Workbooks и Cells, то получаем еще больше типов взаимодействия.

Необязательные параметры и именованные аргументы

Поскольку API-интерфейсы COM не поддерживают перегрузку функций, очень часто приходится иметь дело с функциями, принимающими многочисленные параметры, часть которых являются необязательными. Например, вот как можно вызвать метод Save рабочей книги Excel:

```
var missing = System.Reflection.Missing.Value;
workBook.SaveAs (@"d:\temp.xlsx", missing, missing, missing, missing,
    missing, Excel.XlSaveAsAccessMode.xlNoChange, missing, missing,
    missing, missing);
```

Хорошая новость заключается в том, что поддержка необязательных параметров в C# осведомлена о COM, поэтому можно поступать просто так:

```
workBook.SaveAs (@"d:\temp.xlsx");
```

(Как объяснялось в главе 3, необязательные параметры “расширяются” компилятором в полную форму.)

Именованные аргументы позволяют указывать дополнительные аргументы независимо от их позиций:

```
workBook.SaveAs (@"d:\test.xlsx", Password:"foo");
```

Невидимые параметры `ref`

Некоторые API-интерфейсы COM (в частности, Microsoft Word) открывают доступ к функциям, которые объявляют *каждый* параметр как передаваемый по ссылке вне зависимости от того, модифицирует функция его значение или нет. Причина — выигрыш в производительности из-за отсутствия необходимости копировать значения аргументов (хотя *фактический* выигрыш в производительности незначителен).

Исторически сложилось так, что вызов методов подобного рода в коде C# был затруднен, поскольку для каждого аргумента приходилось указывать ключевое слово `ref`, а это препятствовало использованию необязательных параметров. Например, для открытия документа Word раньше нужно было поступать следующим образом:

```
object filename = "foo.doc";
object notUsed1 = Missing.Value;
object notUsed2 = Missing.Value;
object notUsed3 = Missing.Value;

...
Open (ref filename, ref notUsed1, ref notUsed2, ref notUsed3, ...);
```

Благодаря неявным ссылочным параметрам модификатор `ref` в вызовах функций COM можно опускать, делая возможным применение необязательных параметров:

```
word.Open ("foo.doc");
```

Однако следует помнить о том, что если вызываемый метод COM действительно изменит значение аргумента, то никакой ошибки не возникнет — ни на этапе компиляции, ни во время выполнения.

Индексаторы

Возможность не указывать модификатор `ref` дает еще одно преимущество: индексаторы COM с параметрами `ref` становятся доступными через обычный синтаксис индексаторов C#. В противном случае это было бы запрещено, т.к. параметры `ref/out` не поддерживаются индексаторами C#.

Можно также обращаться к свойствам COM, которые принимают аргументы. В следующем примере `Foo` является свойством, принимающим целочисленный аргумент:

```
myComObject.Foo [123] = "Hello";
```

Самостоятельное написание таких свойств в C# все еще запрещено: тип может открывать доступ к индексатору только на самом себе (“стандартный” индексатор). Таким образом, если необходимо написать код C#, который сделал бы предыдущий оператор законным, то свойство `Foo` должно было бы возвращать другой тип, открывающий доступ к (стандартному) индексатору.

Динамическое связывание

Есть два способа, которыми динамическое связывание может помочь при обращении к компонентам COM.

Первый способ связан с разрешением доступа к компоненту COM без типа взаимодействия COM. Для этого необходимо вызвать метод `Type.GetTypeFromProgID` с именем компонента COM, чтобы получить экземпляр COM и затем воспользоваться динамическим связыванием с целью вызова методов данного экземпляра. Естественно, средство IntelliSense здесь недоступно, и проверки на этапе компиляции невозможны:

```
Type excelAppType = Type.GetTypeFromProgID ("Excel.Application", true);
dynamic excel = Activator.CreateInstance (excelAppType);
excel.Visible = true;
dynamic wb = excel.Workbooks.Add();
excel.Cells [1, 1].Value2 = "foo";
```

(Аналогичной цели можно достичь и намного более запутанным путем, применяя рефлексию вместо динамического связывания.)



Вариацией на эту тему является обращение к компоненту COM, который поддерживает только интерфейс `IDispatch`. Тем не менее, такие компоненты встречаются довольно редко.

Динамическое связывание также может быть удобным (в меньшей степени) при работе с COM-типов `variant`. По причинам, обусловленным скорее неудачным проектным решением, нежели необходимости, функции API-интерфейса COM зачастую буквально усыпаны данным типом, который является грубым эквивалентом типа `object` в .NET. Если вы включите параметр `Embed Interop Types` в своем проекте, тогда исполняющая среда будет отображать `variant` на `dynamic` вместо отображения `variant` на `object`, устранив необходимость в приведениях. Например, законно было бы поступить так:

```
excel.Cells [1, 1].Font.FontStyle = "Bold";
```

вместо:

```
var range = (Excel.Range) excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

Недостаток такого способа работы связан с утратой возможности автозавершения, поэтому вы должны точно знать, что свойство по имени `Font` существует. По указанной причине обычно проще динамически присваивать результат известному типу взаимодействия:

```
Excel.Range range = excel.Cells [1, 1];
range.Font.FontStyle = "Bold";
```

Как видите, код не намного короче, чем при подходе в старом стиле!

Отображение `variant` на `dynamic` принято по умолчанию и является функцией включения параметра `Embed Interop Types` (Внедрять типы взаимодействия) для ссылки.

Внедрение типов взаимодействия

Ранее упоминалось о том, что C# обычно обращается к компонентам COM через типы взаимодействия, которые генерируются путем запуска инструмента `tlbimp.exe` (напрямую или через Visual Studio).

По историческим причинам единственным вариантом была ссылка на сборки взаимодействия, как в случае любых других сборок. Дело могло быть хлопотным, потому что сборки взаимодействия для сложных компонентов COM нередко оказываются довольно большими. Скажем, крошечный дополнительный модуль для Microsoft Word требует сборки взаимодействия, которая на порядки больше его самого.

Вместо ссылки на сборку взаимодействия можно внедрять лишь те части, которые используются. Компилятор анализирует сборку на предмет типов и членов, которые требуются в приложении, и внедряет определения (только) таких типов и членов прямо в приложение. В итоге устраняется эффект разбужания кода, а также отпадает необходимость в поставке дополнительного файла.

Щелкните на ссылке COM в проводнике решения Visual Studio и в окне свойств проекта установите параметр Embed Interop Types (Внедрять типы взаимодействия) в true или отредактируйте файл .csproj, как было описано ранее в главе в разделе “Обращение к компоненту COM из C#”.

Эквивалентность типов

Среда CLR поддерживает эквивалентность типов для связанных типов взаимодействия. Это означает, что если две сборки связываются с каким-то типом взаимодействия, то такие типы будут считаться эквивалентными, если они являются оболочками для одного и того же типа COM. Сказанное справедливо, даже когда сборки взаимодействия, с которыми они связаны, были генерированы независимо друг от друга.



Эквивалентность типов полагается на атрибут TypeIdentifier Attribute из пространства имен System.Runtime.InteropServices. Компилятор автоматически применяет его во время связывания со сборками взаимодействия. Затем типы COM считаются эквивалентными, если они имеют одинаковые идентификаторы GUID.

Открытие объектов C# для COM

Существует также возможность написания классов C#, которые могут потребляться в мире COM. Среда CLR делает это возможным через посредника под названием *вызываемая оболочка COM* (COM-callable wrapper — CCW). Оболочка CCW маршализирует типы между двумя мирами (подобно RCW), а также реализует интерфейс IUnknown (и дополнительно IDispatch), как того требует протокол COM. Временем жизни оболочки CCW управляет сторона COM через подсчет ссылок (а не посредством сборщика мусора CLR).

Любой открытый класс можно делать доступным COM (в качестве “внутрипроцессного” сервера). Для начала создайте интерфейс, назначьте ему идентификатор GUID (в Visual Studio можно выбрать пункт меню Tools⇒Create GUID (Сервис⇒Создать GUID)), объявит его видимым COM и установите тип интерфейса:

```
namespace MyCom
{
    [ComVisible(true)]
    [Guid ("226E5561-C68E-4B2B-BD28-25103ABC3B1")] // Измените этот GUID
    [InterfaceType (ComInterfaceType.InterfaceIsIUnknown)]
    public interface IServer
    {
        int Fibonacci();
    }
}
```

Затем сделайте доступной реализацию нашего интерфейса, назначив ей идентификатор GUID:

```
namespace MyCom
{
    [ComVisible(true)]
    [Guid ("09E01FCD-9970-4DB3-B537-0EC555967DD9")]      // Измените этот GUID
    public class Server
    {
        public ulong Fibonacci (ulong whichTerm)
        {
            if (whichTerm < 1) throw new ArgumentException (...);
            ulong a = 0;
            ulong b = 1;
            for (ulong i = 0; i < whichTerm; i++)
            {
                ulong tmp = a;
                a = b;
                b = tmp + b;
            }
            return a;
        }
    }
}
```

Отредактируйте свой файл .csproj, добавив следующую строку (выделенную полужирным):

```
<PropertyGroup>
    <EnableComHosting>true</EnableComHosting>
</PropertyGroup>
```

Теперь при построении проекта генерируется дополнительный файл MyCom.comhost.dll, который можно зарегистрировать для взаимодействия с СОМ. (Имейте в виду, что в зависимости от конфигурации проекта этот файл будет 32- или 64-битным: в таком сценарии не существует понятия “любой центральный процессор”.) Откройте окно командной подсказки с повышенными полномочиями, перейдите в каталог с вашей DLL-библиотекой и введите команду regsvr32 MyCom.comhost.dll.

Затем вы сможете использовать свой компонент СОМ в коде на большинстве языков, способных взаимодействовать с СОМ. Например, создайте в текстовом редакторе показанный ниже сценарий на Visual Basic и запустите его, дважды щелкнув на имени файла в проводнике Windows или введя имя файла в окне командной подсказки, как поступили бы с обычновенной программой:

```
REM Сохраните файл как ComClient.vbs
Dim obj
Set obj = CreateObject ("MyCom.Server")
result = obj.Fibonacci(12)
Wscript.Echo result
```

Обратите внимание, что .NET Framework нельзя загрузить в тот же процесс, что и .NET 5+ или .NET Core. Следовательно, COM-сервер .NET 5+ не может быть загружен в процесс COM-клиента .NET Framework и наоборот.

Включение COM без регистрации

Традиционно COM добавляет информацию о типах в реестр. Для управления активизацией объектов COM без регистрации используется файл манифеста, а не реестр. Чтобы включить данное средство, добавьте в свой файл .csproj приведенную далее строку (выделенную полужирным):

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.0</TargetFramework>
  <EnableComHosting>true</EnableComHosting>
  <EnableRegFreeCom>true</EnableRegFreeCom>
</PropertyGroup>
```

В результате построения проекта будет сгенерирован файл MyCom.X.manifest.



Поддержка генерации библиотеки типов COM (*.tlb) в .NET 5+ отсутствует. Для низкоуровневых объявлений в интерфейсе вы можете вручную написать файл IDL (Interface Definition Language — язык определения интерфейсов) или заголовочный файл C++.



Регулярные выражения

Язык регулярных выражений распознает символьные образцы. Типы .NET, поддерживающие регулярные выражения, основаны на регулярных выражениях Perl 5 и обеспечивают функциональность как поиска, так и поиска/замены.

Регулярные выражения используются для решения следующих задач:

- проверка текстового ввода, такого как пароли и телефонные номера;
- преобразование текстовых данных в более структурированные формы (например, в строку версии NuGet);
- замена образцов текста в документе (например, только целых слов).

Настоящая глава разделена на концептуальные разделы, обучающие основам регулярных выражений в .NET, и справочные разделы, в которых приводится описание языка регулярных выражений.

Все типы для работы с регулярными выражениями определены в пространстве имен `System.Text.RegularExpressions`.



Все примеры, приведенные в главе, можно загрузить вместе с утилитой LINQPad, которая также включает интерактивный инструмент RegEx (для его открытия — нажмите комбинацию клавиш `<Ctrl+Shift+F1>`). Онлайновая версия инструмента доступна по ссылке <http://regexstorm.net/tester>.

Основы регулярных выражений

Одной из наиболее распространенных операций регулярных выражений является *квантификатор*. Операция `?` — это квантификатор, который соответствует предшествующему элементу 0 или 1 раз. Другими словами, `?` означает *необязательный*. Элемент представляет собой либо одиночный символ, либо сложную структуру символов в квадратных скобках. Например, регулярное выражение `"colou?r"` соответствует `color` и `colour`, но не `colouur`:

```
Console.WriteLine (Regex.Match ("color", @"colou?r").Success); // True
Console.WriteLine (Regex.Match ("colour", @"colou?r").Success); // True
Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success); // False
```

Метод `Regex.Match` выполняет поиск внутри большой строки. Возвращаемый им объект имеет свойства для позиции (`Index`) и длины (`Length`) совпадения, а также свойство для действительного значения (`Value`) совпадения:

```
Match m = Regex.Match ("any colour you like", @"colou?r");
Console.WriteLine (m.Success); // True
Console.WriteLine (m.Index); // 4
Console.WriteLine (m.Length); // 6
Console.WriteLine (m.Value); // colour
Console.WriteLine (m.ToString()); // colour
```

Метод `Regex.Match` можно воспринимать как более мощную версию метода `IndexOf` типа `string`. Разница в том, что он ищет совпадение с *образцом*, а не с *литеральной строкой*.

Метод `IsMatch` — сокращение для вызова метода `Match` с последующей проверкой свойства `Success`.

Механизм регулярных выражений по умолчанию работает слева направо, поэтому возвращается только самое левое соответствие. Для возвращения дополнительных совпадений можно применять метод `NextMatch`:

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",
    @"colou?rs?");
Match m2 = m1.NextMatch();
Console.WriteLine (m1); // color
Console.WriteLine (m2); // colours
```

Метод `Matches` возвращает все совпадения в виде массива. Предыдущий пример можно переписать, как показано ниже:

```
foreach (Match m in Regex.Matches
    ("One color? There are two colours in my head!", @"colou?rs?"))
    Console.WriteLine (m);
```

Еще одной распространенной операцией регулярных выражений является *перестановка*, обозначаемая вертикальной чертой, т.е. `|`. Перестановка выражает альтернативы. Следующий код дает совпадения для `Jen`, `Jenny` и `Jennifer`:

```
Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?")); // True
```

Скобки вокруг перестановки отделяют альтернативы от остальной части выражения.



При поиске совпадений с регулярными выражениями можно указывать тайм-аут. Если операция поиска совпадения занимает больше времени, чем заданное в объекте `TimeSpan`, тогда генерируется исключение `RegexMatchTimeoutException`. Прием может быть полезен, когда программа обрабатывает регулярные выражения, предоставляемые пользователем (например, в диалоговом окне расширенного поиска), потому что при этом предотвращается бесконечное зацикливание неправильно сформированных регулярных выражений.

Скомпилированные регулярные выражения

В некоторых рассмотренных ранее примерах мы многократно вызывали статический метод `RegEx` с одним и тем же образцом. В таких случаях альтернативным подходом является создание объекта `Regex` с этим образцом и флагом `RegexOptions.Compiled`, а затем вызов методов экземпляра:

```
Regex r = new Regex(@"sausages?", RegexOptions.Compiled);
Console.WriteLine(r.Match("sausage")); // sausage
Console.WriteLine(r.Match("sausages")); // sausages
```

Флаг `RegexOptions.Compiled` инструктирует экземпляр `RegEx` о том, что должна использоваться облегченная генерация кода (`DynamicMethod` в `Reflection.Emit`) для динамического построения и компиляции кода, настроенного на это конкретное выражение. В результате обеспечивается более быстрое сопоставление за счет затрат на первоначальную компиляцию.

Создать объект `Regex` можно также без применения `RegexOptions.Compiled`. Экземпляр `Regex` является неизменяемым.



Механизм регулярных выражений характеризуется высокой скоростью. Даже без компиляции нахождение простого совпадения требует менее микросекунды.

Перечисление флагов `RegexOptions`

Перечисление флагов `RegexOptions` позволяет настраивать поведение сопоставления. Распространенное применение `RegexOptions` связано с выполнением поиска, нечувствительного к регистру символов:

```
Console.WriteLine(Regex.Match("a", "A", RegexOptions.IgnoreCase)); // a
```

Это задействует правила для эквивалентности регистров символов текущей культуры. Флаг `CultureInvariant` позволяет затребовать инвариантную культуру:

```
Console.WriteLine(Regex.Match("a", "A", RegexOptions.IgnoreCase
| RegexOptions.CultureInvariant));
```

Большинство флагов `RegexOptions` можно также активизировать внутри самого регулярного выражения с использованием однобуквенного кода:

```
Console.WriteLine(Regex.Match("a", @"(?i)A")); // a
```

Действие флагов можно включать и отключать на протяжении всего выражения:

```
Console.WriteLine(Regex.Match("AAa", @"(?i)a(?-i)a")); // Aa
```

Еще одним полезным флагом является `IgnorePatternWhitespace` или `(?x)`. Он позволяет вставлять пробельные символы, чтобы улучшить читабельность регулярного выражения — без трактовки таких символов литеральным образом.

Значение `NonBacktracking` (введенное в .NET 7) заставляет механизм регулярных выражений применять алгоритм сопоставления с продвижением только вперед. Обычно это приводит к снижению производительности и отключению некоторых расширенных функций, таких как просмотр вперед или просмотр назад. Тем не менее, в таком случае также предотвращается почти бесконечное время выполнения искаженных или злонамеренно созданных выражений, уменьшая потенциальную атаку типа отказа в обслуживании при обработке регулярных выражений, которые предоставлены пользователем (атака ReDOS). В сценариях подобного рода также полезно указывать тайм-аут.

В табл. 25.1 приведены все значения `RegexOptions` вместе с их однобуквенными кодами.

Таблица 25.1. Параметры регулярных выражений

Значение перечисления	Код в регулярном выражении	Описание
None		
<code>IgnoreCase</code>	<code>i</code>	Игнорировать регистр символов (по умолчанию регулярные выражения чувствительны к регистру символов)
<code>Multiline</code>	<code>m</code>	Изменить <code>^</code> и <code>\$</code> так, чтобы они соответствовали началу/концу строчки текста, а не началу/концу всей строки
<code>ExplicitCapture</code>	<code>n</code>	Захватывать только явно именованные или явно нумерованные группы (как описано в разделе “Группы” далее в главе)
<code>Compiled</code>		Инициировать компиляцию регулярного выражения в IL (см. раздел “Скомпилированные регулярные выражения” ранее в главе)
<code>Singleline</code>	<code>s</code>	Сделать точку <code>(.)</code> соответствующей любому символу (вместо соответствия любому символу кроме <code>\n</code>)
<code>IgnorePatternWhitespace</code>	<code>x</code>	УстраниТЬ из образца неотмененные пробельные символы
<code>RightToLeft</code>	<code>r</code>	Выполнять поиск справа налево; указывать где-то посередине не разрешено
<code>ECMAScript</code>		Обеспечить совместимость с ECMA (по умолчанию реализация не совместима с ECMA)
<code>CultureInvariant</code>		Отключить поведение, специфичное для культуры, при сравнении строк
<code>NonBacktracking</code>		Отключить возврат назад с целью обеспечения предсказуемой (хотя и более низкой) производительности

Отмена символов

Регулярные выражения имеют следующие метасимволы, которые трактуются специальным образом, отличающимся от их литерального смысла:

\ * + ? | { [() ^ \$. #

Чтобы применить метасимвол литерально, его потребуется предварить обратной косой чертой, т.е. отменить. В следующем примере мы отменяем символ ? для сопоставления со строкой "what?":

```
Console.WriteLine (Regex.Match ("what?", @"what\?")); // what? (правильно)
Console.WriteLine (Regex.Match ("what?", @"what?")); // what (неправильно)
```



Если символ находится внутри набора (в квадратных скобках), то данное правило не действует, и метасимволы интерпретируются литеральным образом. Наборы обсуждаются в следующем разделе.

Методы Escape и Unescape класса Regex преобразуют строку, содержащую метасимволы регулярных выражений, путем замены их отмененными эквивалентами и наоборот. Например:

```
Console.WriteLine (Regex.Escape ("@\"?")); // \?
Console.WriteLine (Regex.Unescape (@"\?")); // ?>
```

Все строки регулярных выражений в настоящей главе представлены с помощью литерала @ из C#. Так сделано для того, чтобы обойти механизм отмены языка C#, в котором также используется обратная косая черта. Без символа @ литеральная обратная косая черта потребовала бы указания четырех таких символов:

```
Console.WriteLine (Regex.Match ("\\", "\\\\")); // \
```

Если не включена опция (?x), тогда пробелы в регулярных выражениях трактуются литеральным образом:

```
Console.WriteLine (Regex.IsMatch ("hello world", @"hello world")); // True
```

Наборы символов

Наборы символов действуют в качестве групповых символов для отдельного множества символов.

Выражение	Описание	Инверсия ("не")
[abcdef]	Соответствует одиночному символу в списке	[^abcdef]
[a-f]	Соответствует одиночному символу в диапазоне	[^a-f]
\d	Соответствует чему угодно из категории цифр Unicode. В режиме ECMAScript это то же самое, что и [0-9]	\D
\w	Соответствует символу, который допустим в словах (по умолчанию варьируется согласно CultureInfo.CurrentCulture; например, в английском языке это то же самое, что и [a-zA-Z_0-9])	\W

Выражение	Описание	Инверсия ("не")
\s	Соответствует пробельному символу, т.е. любому символу, для которого <code>char.IsWhiteSpace</code> возвращает <code>true</code> (включая пробелы Unicode). В режиме ECMAScript это то же самое, что и <code>[\n\r\t\f\v]</code>	\S
\p{категория}	Соответствует символу в указанной категории	\P
.	(Стандартный режим.) Соответствует любому символу кроме <code>\n</code>	\n
.	(Режим SingleLine.) Соответствует любому символу	\n

Для соответствия в точности одному символу из набора поместите набор символов в квадратные скобки:

```
Console.WriteLine(Regex.Matches("That is that.", "[Tt]hat").Count); // 2
```

Для соответствия любому символу, исключая перечисленные в наборе, поместите набор в квадратные скобки и укажите `^` перед первым символом набора:

```
Console.WriteLine(Regex.Match("quiz qwerty", "q[^aeiou]").Index); // 5
```

С помощью дефиса можно задавать диапазон символов. Следующее выражение соответствует шахматному ходу:

```
Console.WriteLine(Regex.Match("b1-c4", @"[a-h]\d-[a-h]\d").Success); // True
```

`\d` указывает цифровой символ, поэтому `\d` будет соответствовать любой цифре. `\D` соответствует любому нецифровому символу.

`\w` указывает символ, допустимый в словах, что включает буквы, цифры и подчеркивание. `\W` соответствует любому символу, наличие которого в словах не допускается. Это также работает ожидаемым образом и для неанглийских букв, таких как кириллица.

. соответствует любому символу кроме `\n` (но разрешает `\r`).

`\p` соответствует символу в указанной категории, такой как `{Lu}` для буквы верхнего регистра или `{P}` для знака пунктуации (список категорий будет приведен в справочном разделе далее в главе):

```
Console.WriteLine(Regex.IsMatch("Yes, please", @"\p{P}")); // True
```

Мы приведем больше случаев применения `\d`, `\w` и ., когда будем комбинировать их с квантификаторами.

Квантификаторы

Квантификаторы обеспечивают соответствие элементу указанное количество раз.

Квантификатор	Описание
*	Ноль или больше совпадений
+	Одно или больше совпадений
?	Ноль или одно совпадение
{n}	В точности n совпадений
{n,}	По меньшей мере, n совпадений
{n,m}	Количество совпадений между n и m

Квантификатор * обеспечивает соответствие предшествующего символа или группы ноль или более раз. Следующее выражение соответствует имени файла cv.docx, а также любым версиям имени с числами (например, cv2.docx, cv15.docx):

```
Console.WriteLine (Regex.Match ("cv15.docx", @"cv\d*\.\.docx").Success); // True
```

Обратите внимание, что мы должны отменить символ точки в расширении файла с помощью обратной косой черты.

Показанное ниже выражение допускает наличие любых символов между cv и .docx и эквивалентно команде dir cv*.docx:

```
Console.WriteLine (Regex.Match ("cvjoint.docx", @"cv.*\.\.docx").Success); // True
```

Квантификатор + обеспечивает соответствие предшествующего символа или группы один или более раз, например:

```
Console.WriteLine (Regex.Matches ("slow! yeah slooow!", "slo+w").Count); // 2
```

Квантификатор {} обеспечивает соответствие указанному количеству (или диапазону) повторений. Следующее выражение выводит показания артериального давления:

```
Regex bp = new Regex (@"\d{2,3}/\d{2,3}");
Console.WriteLine (bp.Match ("It used to be 160/110")); // 160/110
Console.WriteLine (bp.Match ("Now it's only 115/75")); // 115/75
```

Жадные или ленивые квантификаторы

По умолчанию квантификаторы являются **жадными** как противоположность **ленивым** квантификаторам. Жадный квантификатор повторяется настолько много раз, сколько может, прежде чем продолжить. Ленивый квантификаторы повторяются настолько мало раз, сколько может, прежде чем продолжить. Для того чтобы сделать любой квантификатор ленивым, его необходимо снабдить суффиксом в виде символа ?. Чтобы проиллюстрировать разницу, рассмотрим следующий фрагмент HTML-разметки:

```
string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";
```

Предположим, что нужно извлечь две фразы, выделенные курсивом. Если мы запустим следующий код:

```
foreach (Match m in Regex.Matches (html, @"<i>.*</i>"))
    Console.WriteLine (m);
```

то результатом будет не два, а одно совпадение:

```
<i>By default</i> quantifiers are <i>greedy</i>
```

Проблема в том, что квантификатор `*` жадным образом повторяется настолько много раз, сколько может, перед обнаружением соответствия `</i>`. Таким образом, он поглощает первое вхождение `</i>`, останавливаясь только на финальном вхождении `</i>` (последнее место, где все еще обеспечивается совпадение).

Если сделать квантификатор ленивым, тогда он остановится в *первом* месте, после которого остаток выражения может дать совпадение:

```
foreach (Match m in Regex.Matches (html, @"<i>.*?</i>"))
    Console.WriteLine (m);
```

Вот результат:

```
<i>By default</i>
<i>greedy</i>
```

Утверждения нулевой ширины

Язык регулярных выражений позволяет размещать условия, которые должны удовлетворяться *до* или *после* совпадения, через *просмотр назад*, *просмотр вперед*, *привязки* и *границы слов*. Все вместе они называются *утверждениями нулевой ширины*, потому что они не увеличивают ширину (или длину) самого совпадения.

Просмотр вперед и просмотр назад

Конструкция `(?=expr)` проверяет, соответствует ли следующий за ней текст выражению `expr`, не включая `expr` в результат. Это называется *положительным просмотром вперед*. В приведенном ниже примере мы ищем число, за которым расположено слово `miles`:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)"));
```

Вот вывод:

```
25
```

Обратите внимание, что слово `miles` не возвращается как часть результата, хотя оно требовалось для того, чтобы удовлетворить условие совпадения.

После успешного просмотра вперед поиск совпадения продолжается, как если бы предварительный просмотр никогда не выполнялся. Таким образом, если добавить к выражению конструкцию `.*`, как показано ниже:

```
Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles) .*"));
то результатом будет 25 miles more.
```

Просмотр вперед может быть полезен для навязывания правил выбора сильных паролей. Предположим, что пароль должен иметь длину не менее шести символов и содержать, по крайней мере, одну цифру. С помощью просмотра задачу можно решить следующим образом:

```
string password = "...";
bool ok = Regex.IsMatch (password, @"^(?=.*\d).{6,}");
```

Здесь сначала осуществляется *просмотр вперед*, чтобы удостовериться в наличии цифры где-нибудь в строке. Если цифра обнаружена, тогда происходит возврат к позиции перед началом предварительного просмотра и производится проверка соответствия шести или более символам. (В разделе “Рецептурный справочник по регулярным выражениям” далее в главе мы приводим более существенный пример проверки паролей.)

Противоположностью является конструкция *отрицательного просмотра вперед*, т.е. `(?!expr)`. Она требует, чтобы совпадение *не* следовало за выражением `expr`. Приведенное далее выражение соответствует `good`, если только *позже* в строке не встречается `however` или `but`:

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine (Regex.IsMatch ("Good work! But...", regex)); // False
Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex)); // True
```

Конструкция `(?<=expr)` обозначает *положительный просмотр назад* и требует, чтобы совпадению *предшествовало* указанное выражение. Противоположная конструкция, `(?<!expr)`, обозначает *отрицательный просмотр назад* и требует, чтобы совпадению *не предшествовало* указанное выражение. Например, следующее выражение соответствует `good`, если только `however` не встречалось *ранее* в строке:

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine (Regex.IsMatch ("However good, we...", regex)); // False
Console.WriteLine (Regex.IsMatch ("Very good, thanks!", regex)); // True
```

Приведенные примеры можно было бы усовершенствовать за счет добавления *утверждений границ слов*, которые вскоре будут описаны.

Привязки

Привязки `^` и `$` соответствуют конкретной позиции. По умолчанию:

- `^` соответствует *началу строки*;
- `$` соответствует *концу строки*.



В зависимости от контекста символ `^` означает *привязку* или *отрицание класса символов*.

В зависимости от контекста символ `$` означает *привязку* или *маркер группы замены*.

Например:

```
Console.WriteLine (Regex.Match ("Not now", "^[Nn]o")); // No
Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$")); // F
```

Если указать `RegexOptions.Multiline` или включить в выражение конструкцию `(?m)`, то:

- символ `^` соответствует *началу всей строки* или *строки текста* (сразу после `\n`);
- символ `$` соответствует *концу всей строки* или *строки текста* (непосредственно перед `\n`).

С использованием символа \$ в многострочном (Multiline) режиме связана одна загвоздка: новая строка в Windows почти всегда обозначается с помощью комбинации \r\n, а не просто \n. Это значит, что для обеспечения полезности символа \$ в случае файлов Windows обычно придется искать совпадение также и с \r, применяя *положительный просмотр вперед*:

```
(?=\\r?\\$)
```

Положительный просмотр вперед гарантирует, что \r не станет частью результата. Показанный ниже код соответствует строкам, которые заканчиваются на ".txt":

```
string fileNames = "a.txt" + "\\r\\n" + "b.doc" + "\\r\\n" + "c.txt";
string r = @".+\\.txt(?=\\r?\\$)";
foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline))
    Console.WriteLine (m + " ");
```

Вывод:

```
a.txt c.txt
```

Следующий код соответствует всем пустым строкам текста внутри строки s:

```
MatchCollection emptyLines = Regex.Matches (s, "^(?=\\r?\\$)", RegexOptions.Multiline);
```

Показанный далее код соответствует всем строкам текста, которые либо пусты, либо содержат только пробельные символы:

```
MatchCollection blankLines = Regex.Matches (s, "^[ \\t]*(?=\\r?\\$)", RegexOptions.Multiline);
```



Поскольку привязка соответствует позиции, а не символу, указание одной лишь привязки соответствует пустой строке:

```
Console.WriteLine (Regex.Match ("x", "\\$").Length); // 0
```

Границы слов

Утверждение границы слова \b дает совпадение, когда символы, допустимые в словах (\w), соседствуют с:

- символами, не допустимыми в словах (\W);
- началом/концом строки (^ и \$).

\b часто используется для соответствия целым словам:

```
foreach (Match m in Regex.Matches ("Wedding in Sarajevo", @"\b\w+\b"))
    Console.WriteLine (m);
```

Вывод:

```
Wedding
in
Sarajevo
```

Следующие операторы подчеркивают эффект от границы слова:

```
int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count; // 1
int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count; // 2
```

В приведенном далее выражении применяется положительный просмотр вперед для возврата слов, за которыми следуют символы (sic):

```
string text = "Don't loose (sic) your cool";
Console.WriteLine(Regex.Match(text, @"\b\w+\b\s(=?\1(sic\1))")); // loose
```

Группы

Временами регулярное выражение удобно разделять на последовательности подвыражений, или *группы*. Например, рассмотрим следующее регулярное выражение, которое представляет телефонные номера в США, такие как 206-465-1918:

```
\d{3}-\d{3}-\d{4}
```

Предположим, что мы хотим разделить его на две группы: код зоны и локальный номер. Задачу можно решить, используя круглые скобки для захвата каждой группы:

```
(\d{3})-(\d{3}-\d{4})
```

Затем группы можно извлекать программно:

```
Match m = Regex.Match("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");
Console.WriteLine(m.Groups[1]); // 206
Console.WriteLine(m.Groups[2]); // 465-1918
```

Нулевая группа представляет полное совпадение. Другими словами, она имеет то же самое значение, что и свойство Value совпадения:

```
Console.WriteLine(m.Groups[0]); // 206-465-1918
Console.WriteLine(m); // 206-465-1918
```

Группы являются частью самого языка регулярных выражений. Это означает, что вы можете ссылаться на группу внутри регулярного выражения. Синтаксис \n позволяет индексировать группу по ее номеру n в рамках выражения. Например, выражение (\w)ee\1 дает совпадения для deed и reer. В следующем примере мы ищем в строке все слова, начинающиеся и заканчивающиеся на ту же самую букву:

```
foreach (Match m in Regex.Matches("pop pope peep", @"\b(\w)\w+\1\b"))
    Console.WriteLine(m + " "); // pop peep
```

Скобки вокруг \w указывают механизму регулярных выражений на необходимость сохранения подсовпадений в группе (одиночной буквы в данном случае), поэтому их можно будет применять позже. В дальнейшем на данную группу можно ссылаться с использованием \1, что означает первую группу в выражении.

Именованные группы

В длинном или сложном выражении работать с группами удобнее по именам, а не по индексам. Ниже приведен переписанный предыдущий пример, в котором применяется группа по имени 'letter':

```

string regEx =
    @"^b" + // граница слова
    @"(?'letter'\w)" + // соответствует первой букве;
                        // назовем группу 'letter'
    @"\w+" + // соответствует промежуточным буквам
    @"k'letter'" + // соответствует последней букве,
                    // отмеченной как 'letter'
    @"b"; // граница слова

foreach (Match m in Regex.Matches ("bob pope peep", regEx))
    Console.WriteLine (m + " "); // bob peep

```

Вот как назначить имя захваченной группе:

(?'имя-группы' выражение-группы) или (?<имя-группы> выражение-группы)

А вот как ссылаться на группу:

\k'имя-группы' или \k<имя-группы>

В следующем примере производится сопоставление для простого (не вложенного) элемента XML/HTML за счет поиска начального и конечного узлов с совпадающими именами:

```

string regFind =
    @"<(?'tag'\w+?) .*>" + // соответствует первому дескриптору;
                            // назовем группу 'tag'
    @"(?'text'.*)" + // соответствует текстовому содержимому;
                      // назовем группу 'text'
    @"</\k'tag'>"; // соответствует последнему дескриптору,
                     // отмеченному как 'tag'

Match m = Regex.Match ("<h1>hello</h1>", regFind);
Console.WriteLine (m.Groups ["tag"]); // h1
Console.WriteLine (m.Groups ["text"]); // hello

```

Анализ всех возможных вариаций в структуре XML, таких как вложенные элементы, является более сложным. Механизм регулярных выражений .NET имеет расширение под названием “соответствующие сбалансированные конструкции”, которое может помочь в обработке вложенных дескрипторов — информация о нем доступна в Интернете, а также в книге Джеки Фридла *Mastering Regular Expressions* (O'Reilly).

Замена и разделение текста

Метод `RegEx.Replace` работает подобно `string.Replace` за исключением того, что использует регулярное выражение.

Следующий код заменяет строку `cat` строкой `dog`.

В отличие от `string.Replace` слово `catapult` не будет изменено на `dogapult`, потому что совпадения ищутся по границам слов:

```

string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));

```

Вывод:

catapult the dog

Строка замены может ссылаться на исходное совпадение посредством подстановочной конструкции \$0. В следующем примере числа внутри строки помещаются в угловые скобки:

```
string text = "10 plus 20 makes 30";
Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));
```

Вывод:

```
<10> plus <20> makes <30>
```

Обращаться к захваченным группам можно с помощью конструкций \$1, \$2, \$3 и т.д. или \${имя} для именованных групп. Чтобы продемонстрировать, когда это может быть удобно, вспомним регулярное выражение из предыдущего раздела, соответствующее простому элементу XML. За счет перестановки групп мы можем сформировать выражение замены, которое перемещает содержимое элемента в атрибут XML:

```
string regFind =
    @"<(?'tag'\w+?) .*>" +      // соответствует первому дескриптору;
                                    // назовем группу 'tag'
    @"(?'text'.*)" +             // соответствует текстовому содержимому;
                                    // назовем группу 'text'
    @"</\k'tag'>";           // соответствует последнему дескриптору,
                                    // отмеченному как 'tag'

string regReplace =
    @"${tag}" +                // <tag
    @"value="" " +              // value="
    @"${text}" +                // text
    @"""/>";                  // "/>

Console.WriteLine (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));
```

Вот результат:

```
<msg value="hello"/>
```

Делегат MatchEvaluator

Метод Replace имеет перегруженную версию, принимающую делегат MatchEvaluator, который вызывается для каждого совпадения. Это позволяет поручить построение содержимого строки замены коду C#, если язык регулярных выражений в такой ситуации оказывается недостаточно выразительным. Например:

```
Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",
    m => (int.Parse (m.Value) * 10).ToString ()) );
```

Вывод:

```
50 is less than 100
```

В рецептурном справочнике мы покажем, как применять MatchEvaluator с целью защиты символов Unicode специально для HTML-разметки.

Разделение текста

Статический метод `Regex.Split` представляет собой более мощную версию метода `string.Split` с регулярным выражением, обозначающим образец разделителя. В следующем примере мы разделяем строку, в которой разделителем считается любая цифра:

```
foreach (string s in Regex.Split ("a5b7c", @"\d"))
    Console.WriteLine (s + " "); // a b c
```

Результат не содержит сами разделители. Тем не менее, включить разделители можно, поместив выражение внутрь *положительного просмотра вперед*. Следующий код разбивает строку в верблюжьем стиле на отдельные слова:

```
foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))
    Console.WriteLine (s + " "); // one Two Three
```

Рецептурный справочник по регулярным выражениям

Рецепты

Соответствие номеру карточки социального страхования или телефонному номеру в США

```
string ssNum = @"\d{3}-\d{2}-\d{4}";
Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum)); // True
string phone = @"(?x)
    (\d{3}[-\s] | (\d{3})\s? )
    \d{3}[-\s]?
    \d{4}";
Console.WriteLine (Regex.IsMatch ("123-456-7890", phone)); // True
Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone)); // True
```

Извлечение пар "имя = значение" (по одной в строке текста)

Обратите внимание на использование в самом начале директивы `(?m)`:

```
string r = @"(?m)^[\s*('name'\w+)\s*=\s*('value'.*)\s*(?=\r?\$)";
string text =
    @"id = 3
    secure = true
    timeout = 30";
foreach (Match m in Regex.Matches (text, r))
    Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);
```

Вывод:

```
id is 3 secure is true timeout is 30
```

Проверка сильных паролей

Следующий код проверяет, что пароль состоит минимум из шести символов и включает цифру, символ или знак пунктуации:

```
string r = @"(?x)^(?=.* (\d | \p{P} | \p{S} )).{6,}";  
Console.WriteLine (Regex.IsMatch ("abc12", r)); // False  
Console.WriteLine (Regex.IsMatch ("abcdef", r)); // False  
Console.WriteLine (Regex.IsMatch ("ab88yz", r)); // True
```

Строки текста, содержащие, по крайней мере, 80 символов

```
string r = @"(?m)^.{80,}(?=\\r\\$)";  
string fifty = new string ('x', 50);  
string eighty = new string ('x', 80);  
string text = eighty + "\\r\\n" + fifty + "\\r\\n" + eighty;  
Console.WriteLine (Regex.Matches (text, r).Count); // 2
```

Разбор даты/времени (N/N/N H:M:S AM/PM)

Показанное ниже выражение поддерживает разнообразные числовые форматы дат и работает независимо от того, где указан год — в начале или в конце. Директива (?x) улучшает читабельность, разрешая применение пробельных символов; директива (?i) отключает чувствительность к регистру символов (для необязательного указателя AM/PM). Затем к компонентам совпадения можно обращаться через коллекцию Groups:

```
string r = @"(?x) (?i)  
(\d{1,4}) [./-]  
(\d{1,2}) [./-]  
(\d{1,4}) [\sT]  
(\d+):(\d+):(\d+) \s? (AM|PM)?";  
string text = "01/02/2021 3:30:55 PM";  
foreach (Group g in Regex.Match (text, r).Groups)  
    Console.WriteLine (g.Value + " ");
```

Вывод:

01/02/2021 3:30:55 PM 01 02 2021 3 30 55 PM

(Разумеется, выражение не проверяет корректность даты/времени.)

Соответствие римским числам

```
string r =  
    @"(?i)\bm*"  
    +  
    @"(d?c{0,3}|c[dm])"  
    +  
    @"(l?x{0,3}|x[lc])"  
    +  
    @"(v?i{0,3}|i[vx])"  
    +  
    @"\b";  
Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r)); // True
```

Удаление повторяющихся слов

Здесь мы захватываем именованную группу dupe:

```
string r = @"(?<dupe'\w+)>\W\k'dupe'";
string text = "In the the beginning...";
Console.WriteLine (Regex.Replace (text, r, "${dupe}"));
```

Вывод:

```
In the beginning
```

Подсчет слов

```
string r = @"\b(\w|[-])+\b";
string text = "It's all mumbo-jumbo to me";
Console.WriteLine (Regex.Matches (text, r).Count); // 5
```

Соответствие идентификатору GUID

```
string r =
@"(?i)\b" +
@"[0-9a-fA-F]{8}\b" +
@"[0-9a-fA-F]{4}\b" +
@"[0-9a-fA-F]{4}\b" +
@"[0-9a-fA-F]{4}\b" +
@"[0-9a-fA-F]{12}\b";
```



```
string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine (Regex.Match (text, r).Index); // 12
```

Разбор дескриптора XML/HTML

Класс Regex удобен при разборе фрагментов HTML-разметки — особенно, когда документ может быть сформирован некорректно:

```
string r =
@"<(?<'tag'\w+?)\.*>" + // соответствует первому дескриптору;
                           // назовем группу 'tag'
@"(?<'text'.*?)"       + // соответствует текстовому содержимому;
                           // назовем группу 'text'
@"</\k'tag'>";          // соответствует последнему дескриптору,
                           // отмеченному как 'tag'

string text = "<h1>hello</h1>";
Match m = Regex.Match (text, r);
Console.WriteLine (m.Groups ["tag"]); // h1
Console.WriteLine (m.Groups ["text"]); // hello
```

Разделение на слова в верблюжьем стиле

Решение требует *положительного просмотра вперед*, чтобы включить разделители в верхнем регистре:

```
string r = @"(?=[A-Z]) ";
foreach (string s in Regex.Split ("oneTwoThree", r))
Console.Write (s + " "); // one Two Three
```

Получение допустимого имени файла

```
string input = "My \"good\" <recipes>.txt";
char[] invalidChars = System.IO.Path.GetInvalidFileNameChars();
string invalidString = Regex.Escape (new string (invalidChars));
string valid = Regex.Replace (input, "[" + invalidString + "]", "");
Console.WriteLine (valid);
```

Вывод:

```
My good recipes.txt
```

Защита символов Unicode для HTML

```
string htmlFragment = "&# 2007";
string result = Regex.Replace (htmlFragment, @"\u0080-\uFFFF",
    m => "&" + ((int)m.Value[0]).ToString() + ";");
Console.WriteLine (result);           // &#169; 2007
```

Преобразование символов в строке запроса HTTP

```
string sample = "C%23 rocks";
string result = Regex.Replace (
    sample,
    @"% [0-9a-f][0-9a-f]",
    m => ((char) Convert.ToByte (m.Value.Substring (1), 16)).ToString(),
    RegexOptions.IgnoreCase
);
Console.WriteLine (result);           // C# rocks
```

Разбор поисковых терминов Google из журнала веб-статистики

Это должно использоваться в сочетании с предыдущим примером преобразования символов в строке запроса:

```
string sample =
    "http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";
Match m = Regex.Match (sample, @"(?=<google\..+search\?.*q=).+?(?=(&|$))");
string[] keywords = m.Value.Split (
    new[] { '+' }, StringSplitOptions.RemoveEmptyEntries);
foreach (string keyword in keywords)
    Console.Write (keyword + " ");    // greedy quantifiers regex
```

Справочник по языку регулярных выражений

В табл. 25.2–25.12 представлена сводка по грамматике и синтаксису регулярных выражений, которые поддерживаются в реализации .NET.

Таблица 25.2. Управляющие символы

Управляющая последовательность	Описание	Шестнадцатеричный эквивалент
\a	Звуковой сигнал	\u0007
\b	Забой	\u0008
\t	Табуляция	\u0009
\r	Возврат каретки	\u000A
\v	Вертикальная табуляция	\u000B
\f	Перевод страницы	\u000C
\n	Новая строка	\u000D
\e	Отмена	\u001B
\nnn	ASCII-символ nnn в восьмеричной форме (например, \n052)	
\xnn	ASCII-символ nn в шестнадцатеричной форме (например, \x3F)	
\c1	Управляющий ASCII-символ 1 (например, \cG для <Ctrl+G>)	
\unnnn	Unicode-символ nnnn в шестнадцатеричной форме (например, \u07DE)	
\символ	Непреобразуемый символ	

Специальный случай: внутри регулярного выражения комбинация \b означает границу слова за исключением ситуации, когда находится в наборе [], где \b означает символ забоя.

Таблица 25.3. Наборы символов

Выражение	Описание	Инверсия ("не")
[abcdef]	Соответствует одиночному символу в списке	[^abcdef]
[a-f]	Соответствует одиночному символу в диапазоне	[^a-f]
\d	Соответствует десятичной цифре	\D
	То же самое, что и [0-9]	
\w	Соответствует символу, допустимому в сло-вах (по умолчанию варьируется согласно CultureInfo.CurrentCulture; например, в английском языке это то же самое, что и [a-zA-Z_0-9])	\W
\s	Соответствует пробельному символу То же самое, что и [\n\r\t\f\v]	\S
\p{категория}	Соответствует символу в указанной категории (табл. 25.6)	\P
.	(Стандартный режим.) Соответствует любому символу кроме \n	\n
.	(Режим SingleLine.) Соответствует любому символу	\n

Таблица 25.4. Категории символов

Категория	Описание
\p{L}	Буквы
\p{Lu}	Буквы в верхнем регистре
\p{Ll}	Буквы в нижнем регистре
\p{N}	Числа
\p{P}	Знаки пунктуации
\p{M}	Диакритические знаки
\p{S}	Символы
\p{Z}	Разделители
\p{C}	Управляющие символы

Таблица 25.5. Квантификаторы

Квантификатор	Описание
*	Ноль или больше совпадений
+	Одно или больше совпадений
?	Ноль или одно совпадение
{n}	В точности n совпадений
{n, }	По меньшей мере, n совпадений
{n, m}	Количество совпадений между n и m

К любому квантификатору можно применить суффикс ?, чтобы сделать его ленивым, а не жадным.

Таблица 25.6. Подстановки

Выражение	Описание
\$0	Подстановка совпадающего текста
\$номер-группы	Подстановка индексированного номера группы внутри совпадающего текста
\${имя-группы}	Подстановка текстового имени группы внутри совпадающего текста

Подстановки указываются только внутри образца замены.

Таблица 25.7. Утверждения нулевой ширины

Выражение	Описание
^	Начало строки (или строчки текста в многострочном режиме)
\$	Конец строки (или строчки текста в многострочном режиме)
\A	Начало строки (многострочный режим игнорируется)
\z	Конец строки (многострочный режим игнорируется)
\Z	Конец строчки текста или всей строки
\G	Место начала поиска
\b	На границе слова
\B	Не на границе слова
(?=expr)	Продолжать поиск совпадения, только если выражение expr дает совпадение справа (<i>положительный просмотр вперед</i>)
(?!expr)	Продолжать поиск совпадения, только если выражение expr не дает совпадение справа (<i>отрицательный просмотр вперед</i>)
(?<=expr)	Продолжать поиск совпадения, только если выражение expr дает совпадение слева (<i>положительный просмотр назад</i>)
(?<!expr)	Продолжать поиск совпадения, только если выражение expr не дает совпадение слева (<i>отрицательный просмотр назад</i>)
(?>expr)	Подвыражение expr дает совпадение один раз, и не было возврата назад

Таблица 25.8. Конструкции группирования

Синтаксис	Описание
(expr)	Захват давшего совпадение выражения expr в индексированную группу
(?номер)	Захват совпадающей подстроки в группу с указанным номером
(?'имя')	Захват совпадающей подстроки в группу с указанным именем
(?'имя1-имя2')	Отменить определение имя2 и сохранить интервал и текущую группу в имя1; если имя2 не определено, тогда поиск совпадения возвращается назад
(?:expr)	Незахватываемая группа

Таблица 25.9. Обратные ссылки

Синтаксис	Описание
\index	Ссылка на предыдущую захваченную группу по индексу
\k<имя>	Ссылка на предыдущую захваченную группу по имени

Таблица 25.10. Перестановки

Синтаксис	Описание
	Логическое “ИЛИ”
(? (expr) yes no)	Соответствует yes, если выражение expr дает совпадение; в противном случае соответствует по (конструкция по является необязательной)
(? (лате) yes no)	Соответствует yes, если именованная группа лате имеет совпадение; в противном случае соответствует по (конструкция по является необязательной)

Таблица 25.11. Вспомогательные конструкции

Синтаксис	Описание
(?#комментарий)	Встроенный комментарий
#комментарий	Комментарий до конца строки (работает только в режиме IgnorePatternWhitespace)

Таблица 25.12. Параметры регулярных выражений

Параметр	Описание
(?i)	Соответствие, нечувствительное к регистру символов (регистр символов “игнорируется”)
(?m)	Многострочный режим; изменяет ^ и \$ так, что они соответствуют началу и концу любой строки текста
(?n)	Захватывает только явно именованные или пронумерованные группы
(?c)	Компилирует в IL
(?s)	Однострочный режим; изменяет значение точки (.) так, что она соответствует любому символу
(?x)	Устраняет из образца ненужные пробельные символы
(?r)	Поиск справа налево; не может быть указан где-то посередине

Предметный указатель

А

Адаптер
двоичный, 771
потоков, 765; 772
текстовый, 766
Алгоритм
Рэндала, 939
сжатия Brotli, 774
хеширования, 937
Аннотации, 596
Аргумент, 68
Архитектура
исполняющей среды, 38
потоковая, 749
сетевая, 797; 798
Асинхронность, 324
Атомарность, 955
Атрибут, 282; 324
извлечение атрибутов во время
выполнения, 893
псевдоспециальные, 890
специальный, 890
Аутентификация, 808
через заголовки, 809

Б

Балансировка нагрузка, 1007
Барьерь, 982
Безопасность, 789
Безопасность к типам
во время выполнения, 36
на этапе компиляции, 36
Библиотека
базовых классов, 38
базовых классов (BCL), 39
параллельных задач, 993
Avalonia, 41
BCL, 322
.NET BCL, 442
Блокирование, 687; 955; 959
асинхронное, 745
вложенное, 955
монопольное, 950
немонопольное, 950; 965
с двойным контролем, 984
Блокировка, 690; 953
объектов чтения/записи, 968
рекурсия блокировок, 972
Блок операторов, 68; 226

В

Ввод-вывод, 324; 687; 749
файловый, 792
Ветвление, 899
Взаимоблокировка, 956
Виртуализация, 792
Выполнение, 495
отложенное, 509
Выражение, 114
запросов, 36; 472
коллекции, 1042
первичное, 115
пустое, 115
регулярное, 1077
справочник по языку регулярных
выражений, 1093

Г

Генератор
дополнительный, 533
исходного кода, 162
Глобализация, 375
Группа, 1087
именованная, 1087
Группирование, 550

Д

Данные
потоки данных, 324; 749
Декартово произведение, 534
Деконструктор, 60; 146
Делегат, 36; 211
групповой, 213
Дерево выражения, 513; 514
Дескриптор
ожидания событий, 973
Диапазон, 53; 99
Дизассемблер, 916
Динамическое связывание, 63; 287; 1072
Директива
global using, 133
using, 68; 133
using static, 134
Директивы препроцессора, 310
Диспетчеризация
множественная, 927
одиночная, 927

З

Загрузка
ленивая, 508
Задача, 745
комбинаторы задач, 741

отмена задач, 1021
параллелизм задач, 1018
Замыкание, 228
Запись, 36; 51; 156; 265; 753
Запрос, 323; 576
 внешний, 483
 интерпретируемый, 492
 упаковка запросов, 488
Зацикливание, 687
Зондирование
 стандартное, 850

И

Идентификатор, 70
 URI, 801
Изменение
 неразрушающее, 451
Имя
 сокрытие имен, 135
 типа, 868
Индекс, 53; 99
Индексаторы, 154; 1072
Индексация, 422
Инициализатор
 индекса, 62
 модуля, 159
 объекта, 64; 147
 свойства, 151
Инициализация
 ленивая, 982
Инкапсуляция, 35
Интерполяция строк, 96
Интерфейс, 35; 184; 188; 869
 COM, 1068
 EqualityComparer, 455
 ICollection, 416
 ICollection<T>, 416
 IComparable, 398
 IComparer, 457
 IDictionary, 439
 IDictionary< TKey, TValue >, 437
 IDisposable, 410
 IEnumerator, 408
 IEnumerator< T >, 410
 IEqualityComparer, 455
 IList, 417
 IList< T >, 417
 многоплатформенных приложений
 (MAUI), 41
Инфраструктура
 PLINQ, 1006
 WPF, 328

Исключение, 704
 генерация исключений, 238
 обработка исключений, 693; 901
 отправка исключений, 731
 специфических типов, 235
Исполняющая среда, 39
Итератор, 244; 245; 412

К

Канал
 анонимный, 760; 763
 именованный, 760
Квантификатор, 566; 1077; 1082
 жадный, 1083
 ленивый, 1083
Класс, 139
 абстрактный, 169
 атрибутов, 283
 маршализация классов, 1054
 словарей, 438
 статический, 160
 AggregateException, 1029
 ArrayList, 428
 Collection< T >, 412
 Comparer, 457
 Dictionary, 439; 440
 HybridDictionary, 442
 ListDictionary, 442
 OrderedDictionary, 441
Клиент
 обогащенный, 326
 тонкий, 326
Ключевое слово, 70
 контекстное, 71
 await, 716
 base, 170
 into, 487
 let, 491
 public, 75
 stackalloc, 305
Ключ сортировки, 470
Ковариантность, 205; 207; 218
Код
 динамическая генерация кода, 894
Кодировка
 текста, 341
 символов, 769
 UTF-8, 342
 UTF-16, 342; 770
 UTF-32, 342
Коллекция, 322; 407
 выражения коллекций, 1042
 замороженная, 453
 настраиваемая, 444
 неизменяемая, 450

Комбинаторы
задач, 741
специальные, 743

Комментарии, 72

Компаратор, 549
эквивалентности, 436

Компилятор
C#, 68
Roslyn, 326

Компиляция, 68
оперативной (JIT), 38
ранняя (AOT), 38
условная, 659

Конструирование, 422

Конструктор, 171; 908
копирования, 270
неоткрытый, 145
неявный, 145
основной, 155; 274
перегрузка конструкторов, 144
стандартный, 179
статический, 159
экземпляров, 144

Контравариантность, 208; 218

Конфигурирование подключения, 500

Копирование, 427

Кортежи, 60; 260
деконструирование кортежей, 264

Криптография, 325; 935

Куча, 103; 104

Л

Литерал, 68; 71
вещественный, 84
двоичный, 58
необработанный строковый, 45
целочисленный, 84

Локальные переменные
неявно типизированные, 64

Лямбда-выражение, 36; 64; 226; 468; 489; 692
асинхронное, 727
статическое, 229

М

Манифест
приложения, 791; 825
сборки, 824

Маркер, 617

Маршализация
классов, 1054
параметров, 1056
типов, 1052

Массив, 98; 206
встроенный, 45
зубчатый, 101
многомерный, 100

Метаданные, 38; 865

Метка, 130

Метод, 36; 142; 161
анонимный, 232
виртуальный, 444
деконструирующий, 146
локальный, 59; 142
обобщенный, 198; 884
перегрузка методов, 143
преобразования, 555
расширяющий, 64; 256
сжатый до выражения, 142
частичный
расширенный, 162

экземпляра, 213

AsEnumerable, 497

AsQueryable, 512

AsTask, 730

BinarySearch, 424

ConstrainedCopy, 427

Contains, 566

DefineMethod, 905

Element, 578

Empty, 567

FindAll, 425

FromAsync, 746

GetKeyForItem, 447

IndexOf, 425

LastIndexOf, 425

Nodes, 576

Range, 568

Repeat, 568

SetValue, 581

UnionBy, 555

UnionWith, 436

Многопоточность
расширенная, 325; 949

Модель
конфигурирование, 500
DOM-, 569
выражения, 513

Модификатор
асинхронного кода, 142
доступа, 142; 181
наследования, 142
неуправляемого кода, 142
обязательный, 878
статический, 142
частичного метода, 142
in, 109
out, 107
params, 109
ref, 107

Модуль, 826
инициализатор модуля, 159
Мультисловарь, 544

H

Набор, 428
символов, 341; 1081

Навигация, 576

Нarezание, 1040

Наследование, 35; 163; 171

O

Обобщения, 196

Общеязыковая исполняющая среда, 37; 38

Объект

блокировки объектов чтения/записи, 968
выталкивание объекта из стека, 174
заталкивание объекта в стек, 174
инициализаторы объектов, 147
корневой, 639
отслеживание объектов, 504
ожидания (awaiter), 706
Lookup, 544

Ограничение, 201

базового класса, 201
интерфейса, 201
конструктора без параметров, 202
неуправляемого кода, 202

Округление, 372

Оператор, 121

верхнего уровня, 50; 69
выбора, 123
выражения, 122
итераций, 128
объявления, 122
смешанный, 131
перехода, 130
foreach, 410
try, 233
fixed, 304

goto, 130

lock, 951

return, 131

throw, 131

using, 131; 631

Операция, 71; 114

арифметическая, 86

ассоциативность операций, 115

запроса, 464

левоассоциативная, 116

над множествами, 554

над элементами, 559

перегрузка операций, 296

полиморфная, 301

правоассоциативная, 116
приоритеты операций, 115
сравнения, 92
указателя на член, 304
условная (тернарная), 92; 93
эквивалентности, 276

Aggregate, 563

All, 567

Any, 566

AsEnumerable, 558

AsQueryable, 558

C#, 117

Concat, 554

Contains, 566

DefaultIfEmpty, 561

Distinct, 525

ElementAt, 560

Except, 555

GroupBy, 550

GroupJoin, 543

Intersect, 555

Join, 540; 545

LINQ, 517

MinBy, 560

MaxBy, 560

OrderBy, 548

SequenceEqual, 567

SelectMany, 535

ToArray, 558

ToDictionary, 558

ToHashSet, 558

ToList, 558

ToLookup, 558

Where, 522

Zip, 547

Оптимизация, 732

Очереди, 428

П

Память

совместно используемая, 793; 1061

совместно используемая процессами, 794

Параллелизм, 324; 726

задач, 1018

Параметр, 103; 106

атрибута, 283

маршализация параметров, 1056

необязательный, 110

совместимость параметров, 218

Перегрузка, 174

конструкторов, 144

операций, 296; 298

Переменная, 103
генерация локальных переменных, 898
диапазона, 474
захваченные, 228; 478
локальные ссылочные, 58
шаблонные, 59
Перечисление, 191; 382; 385; 408; 423
преобразование перечислений, 192
BindingFlags, 883
Перечислитель, 242
однонаправленный, 1046
Планировщик задач, 1027
Платформа UWP, 330
Подзапрос, 482; 485; 489
Подключение
конфигурирование, 500
Подписание, 945
Подписчики, 219
Подпись Authenticode, 831
Поиск, 424; 754
Поле, 139
очистка полей, 635
Полиморфизм, 35; 164
статический, 190; 300
Последовательность, 463
Посредник, 444; 449
Построители, 452
Поток, 684; 701
адаптер потоков, 772
адаптеры потоков, 765
асинхронные потоки, 728
барьер выполнения потоков, 981
безопасность потоков, 690; 959
вызывающий, 957
данных
асинхронный, 57
использование потоков, 751
локальное хранилище потока, 985
с декораторами, 750
с опорными хранилищами, 750
создание потока, 684
со сжатием, 773
Предикат, 468
Преобразование, 377
булевское, 91
динамическое, 292
между типами с плавающей точкой, 85; 86
между целочисленными типами, 85
распаковывающее, 166
символьное, 94
специальное, 166; 204
ссылочное, 164; 176
упаковывающее, 176
числовое, 176

Преобразователи типов, 374
Предфикс, 591; 595; 608
Приведение
вверх, 164
вниз, 165
Привязки, 1085
Приложение
манифест приложения, 825
UWP, 329
Присваивание
деконструирующее, 147
Программирование
асинхронное, 712
динамическое, 325; 921
параллельное, 325; 993
функциональное, 266
Продолжение, 705; 1024
асинхронное, 63
Проектирование
в конкретные типы, 530
индексированное, 527
Производительность, 957
Прокси-сервер, 807
Пространства имен, 67; 132; 590; 608
с областью видимости на уровне файлов, 48
Протокол
BitTorrent, 817
TCP, 817
Пул потоков, 699

P

Распаковка (unboxing), 175
Распознавание, 174
Реализация
явная, 185
Реентерабельность, 720
Рекурсия блокировок, 972
Ретранслятор, 219
Рефакторинг, 68
Рефлексия, 39; 324; 865; 873
сборок, 888

C

Сборка (assembly), 38, 68; 324; 823; 895
дружественная, 183
загрузка сборок, 844
имена сборок, 829
манифест сборки, 824
мусора
автоматическая, 637
однофайловая, 826
распознавание сборок, 846
рефлексия сборок, 888
ссылочная, 321
portable executable (PE), 823

Сбрасывание, 755
Свойства, 36; 149
автоматические, 65; 151
вычисляемые, 150
допускающие только инициализацию, 878
инициализаторы свойств, 151
скатые до выражений, 151
только для чтения, 150
Связывание
динамическое, 63; 287; 865
специальное, 289
статическое, 880
языковое, 290
Семафор, 965
асинхронный, 967
Сериализация, 283; 326; 573
Сжатие, 780
Сигнализация, 950
Сигнатура, 142; 906
Символ, 93
кодировки символов, 769
Синхронизация, 950
Система Authenticode, 832
Словарь, 436
отсортированный, 442
классы словарей, 438
Слой
прикладной, 38; 39
События, 36; 219
Соединение
перекрестное, 534
Сопоставления, 549
Сортировка, 425
Списки, 428
Справочник по языку регулярных выражений, 1093
Сравнение
структурное, 460
Среда
.NET Framework, 41
Средства доступа только для инициализации, 51
Ссылка this, 149
Стек, 103; 174; 428
вычислений, 896
Строка, 93
запроса, 810
интерполяция строк, 62; 96
конструирование строк, 333
объединение строк, 336
пустая (null), 333
разделение строк, 336
сравнение строк, 337
стандартная форматная, 364
UTF-8, 45; 97
Структура, 178
маршализация структур, 1054
ссылочная, 180
типа записей, 49
BigInteger, 378
Complex, 380
Guid, 386
Half, 379
Utf8JsonReader, 618
Счетчик производительности, 673
чтение данных счетчика, 675

T

Таблица
дочерняя, 537
радужная, 939
родительская, 537
Тайм-аут, 755
Таймеры, 653; 989
многопоточные, 990
однопоточные, 992
Текст
кодировка текста, 341
Тестирование, 376
Технология
Blazor, 37
COM, 1068
P/Invoke, 1051
Тип, 35
активизация типов, 866
анонимный, 259; 490
базовый, 372; 869
булевский, 82; 91
вложенный, 195
возвращаемый
ковариантный, 168
делегата, 211
допускающий null, 56; 81; 248
закрытый, 197
имена типов, 868
маршализация типов, 1052
обобщенный, 196
закрытый, 873
объектный, 82
открытый, 197
преобразователи типов, 374
производный, 168
символьный (char), 82; 331
системный, 322
ссылочный, 80
строковый (string), 82; 94; 333
унификация типов, 382
унифицированная система типов, 35
частичный, 161
числовой, 82

Типизация
 статическая, 36
Трассировки, 679

У

Указатель, 303
 на функцию, 52; 308; 1057
 void, 306
Упаковка (boxing), 175; 188
Управляющие последовательности, 93

Ф

Файл
 безопасность файлов, 781
 проекта, 69
.resources, 836; 838
.resx, 837
Tar, 777
ZIP-, 776

Фильтр
 исключений, 62; 236

Фильтрация
 индексированная, 523
Финализаторы, 160; 639
Флаг HideBySig, 906

Формат
 JSON, 616

Форматирование, 358
 смешанное, 361

Функция, 294
 асинхронная, 57; 722
 виртуальная, 167
 запечатывание функций, 170
 скжатая до выражения, 62; 142

Х

Хеширование, 935; 937
 алгоритм хеширования, 937
 паролей, 939

Хранилище
 опорное, 749

Ц

Центр сертификации (CA), 832
Цикл
 do-while, 128
 for, 129
 while, 128
Цифровой сертификат сайта, 946
Цифровая подпись, 947

Ч

Числовые суффиксы, 84
Член
 анонимный вызов членов обобщенного интерфейса, 884

Чтение, 753

Ш

Шаблон, 166
 асинхронный, 736
 на основе событий, 746
константы, 278
кортежа, 279
позиционный, 56
реляционный, 51; 278
свойства, 56; 277; 280
списка, 46; 282
типа, 166; 277
устаревший, 745
APM, 745
Шифрование, 780
 в памяти, 941
 симметричное, 935; 939
 с открытым ключом, 935; 945

Э

Экземпляр, 213
Элемент, 463

Я

Язык
 безопасный к типам, 36
 интегрированных запросов (LINQ), 463
 промежуточный, 38
 строго типизированный, 37
 управляемый, 38
PLINQ, 995

O'REILLY®

C# 12

Справочник

В этом уже ставшем бестселлером руководстве читатель найдет все необходимые ответы на разнообразные вопросы по языку C# 12 или библиотекам .NET 8. Язык C# обладает замечательной гибкостью и широким размахом, но такое непрекращающееся развитие означает, что по-прежнему есть многие вещи, которые предстоит изучить. В соответствии с традициями справочников O'Reilly это основательно обновленное издание будет наилучшим однотомным источником сведений о языке C#, доступным на сегодняшний день.

Организованное вокруг концепций и сценариев использования, новое издание книги снабдит программистов средней и высокой квалификации лаконичным планом получения глубоких знаний по языку C#, среде CLR и основным библиотекам .NET без длинных вступлений и раздутых примеров.

- Освойте все аспекты языка C#, от синтаксиса и переменных до таких сложных тем, как указатели, записи, замыкания и шаблоны
- Тщательно исследуйте LINQ с помощью трех глав, специально посвященных этой теме
- Узнайте о параллелизме и асинхронности, расширенной многопоточной обработке и параллельном программировании
- Научитесь работать с функциональными средствами .NET, включая регулярные выражения, взаимодействие с сетью, сборки, промежутки, криптографию и рефлексию

Категория: программирование

Предмет рассмотрения: C# / Microsoft .NET

Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-907705-47-0

24045



9 785907 705470

"Одна из немногих книг, которые я держу на столе в качестве быстрого справочника."

Скотт Гатри, Microsoft

"Как новички, так и эксперты найдут здесь все новейшие приемы программирования на C#."

Эрик Липперт,
Комитет по стандартам C#

Джозеф Албахари —
создатель LINQPad и автор книг
C# 10 in a Nutshell
и *C# 12 Pocket Reference*.



<http://www.williamspublishing.com>

<http://oreilly.com>