
Оглавление

Предисловие	10
Для кого написана эта книга	11
Почему я написал эту книгу	12
Типографские соглашения	13
Структура книги	13
Благодарности	14
От издательства	15
Глава 1. Конкурентность: общие сведения	16
Знакомство с конкурентностью	16
Введение в асинхронное программирование	19
Введение в параллельное программирование	25
Введение в реактивное программирование (Rx)	30
Введение в Dataflow	32
Введение в многопоточное программирование	35
Коллекции для конкурентных приложений	36
Современная разработка	37
О ключевых технологиях кратко	38
Глава 2. Основы async	39
2.1. Приостановка на заданный период времени	39
2.2. Возвращение завершенных задач	42
2.3. Передача информации о ходе выполнения операции	45
2.4. Ожидание завершения группы задач	47
2.5. Ожидание завершения любой задачи	50
2.6. Обработка задач при завершении	52
2.7. Обход контекста при продолжении	56
2.8. Обработка исключений из методов async Task	57
2.9. Обработка исключений из методов async void	59
2.10. Создание ValueTask	62
2.11. Потребление ValueTask	64

Глава 3. Асинхронные потоки	68
Асинхронные потоки и Task<T>	68
Асинхронные потоки и IEnumerable<T>	69
Асинхронные потоки и Task<IEnumerable<T>>	69
Асинхронные потоки и IObservable<T>	70
Итоги	70
3.1. Создание асинхронных потоков	72
3.2. Потребление асинхронных потоков	75
3.3. Использование LINQ с асинхронными потоками	77
3.4. Асинхронные потоки и отмена	81
Глава 4. Основы параллельного программирования	85
4.1. Параллельная обработка данных	85
4.2. Параллельное агрегирование	88
4.3. Параллельный вызов	90
4.4. Динамический параллелизм	91
4.5. Parallel LINQ	94
Глава 5. Основы Dataflow	97
5.1. Связывание блоков	97
5.2. Распространение ошибок	99
5.3. Удаление связей между блоками	102
5.4. Регулирование блоков	103
5.5. Параллельная обработка с блоками потока данных	104
5.6. Создание собственных блоков	106
Глава 6. Основы System.Reactive	108
6.1. Преобразование событий .NET	109
6.2. Отправка уведомлений контексту	112
6.3. Группировка данных событий с использованием Window и Buffer	115
6.4. Контроль потоков событий посредством регулировки и выборки	118
6.5. Тайм-ауты	120
Глава 7. Тестирование	124
7.1. Модульное тестирование async-методов	125
7.2. Асинхронные методы модульного тестирования, которые не должны проходить	128
7.3. Модульное тестирование методов async void	131
7.4. Модульное тестирование сетей потоков данных	132
7.5. Модульное тестирование наблюдаемых объектов System.Reactive	134
7.6. Модульное тестирование наблюдаемых объектов System.Reactive с использованием имитации планирования	137

Глава 8. Взаимодействие	142
8.1. Асинхронные обертки для «Async»-методов с «Completed»-событиями	142
8.2. Асинхронные обертки для методов «Begin/End»	144
8.3. Асинхронные обертки для чего угодно	146
8.4. Асинхронные обертки для параллельного кода	148
8.5. Асинхронные обертки для наблюдаемых объектов System.Reactive	149
8.6. Наблюдаемые обертки для асинхронного кода в System.Reactive	151
8.7. Асинхронные потоки и сети потоков данных	153
8.8. Наблюдаемые объекты System.Reactive Observables и сети потока данных	156
8.9. Преобразование наблюдаемых объектов System.Reactive в асинхронные потоки	158
Глава 9. Коллекции	162
9.1. Неизменяемые стеки и очереди	164
9.2. Неизменяемые списки	167
9.3. Неизменяемые множества	169
9.4. Неизменяемые словари	172
9.5. Потокбезопасные словари	174
9.6. Блокирующие очереди	177
9.7. Блокирующие стеки и мультимножества	180
9.8. Асинхронные очереди	182
9.9. Регулировка очередей	186
9.10. Выборка в очередях	189
9.11. Асинхронные стеки и мультимножества	191
9.12. Блокирующие/асинхронные очереди	193
Глава 10. Отмена	199
10.1. Выдача запросов на отмену	200
10.2. Реагирование на запросы на отмену посредством периодического опроса	204
10.3. Отмена по тайм-ауту	206
10.4. Отмена аsync-кода	208
10.5. Отмена параллельного кода	209
10.6. Отмена кода System.Reactive	211
10.7. Отмена сетей потоков данных	213
10.8. Внедрение запросов на отмену	215
10.9. Взаимодействие с другими системами отмены	217

Глава 11. ООП, хорошо сочетающееся с функциональным программированием	219
11.1. Асинхронные интерфейсы и наследование	220
11.2. Асинхронное конструирование: фабрики	222
11.3. Асинхронное конструирование: паттерн асинхронной инициализации	224
11.4. Асинхронные свойства	228
11.5. аsync-события	232
11.6. Асинхронное освобождение	236
Глава 12. Синхронизация	240
12.1. Блокировки и команда lock	246
12.2. Блокировки с аsync	249
12.3. Блокирующие сигналы	251
12.4. Асинхронные сигналы	253
12.5. Регулировка	255
Глава 13. Планирование	258
13.1. Планирование работы в пуле потоков	258
13.2. Выполнение кода с помощью планировщика задач	260
13.3. Планирование параллельного кода	263
13.4. Синхронизация потоков данных с помощью планировщиков	264
Глава 14. Сценарии	266
14.1. Инициализация совместных ресурсов	266
14.2. Отложенное вычисление в System.Reactive	270
14.3. Асинхронное связывание данных	272
14.4. Неявное состояние	275
14.5. Идентичный синхронный и асинхронный код	278
14.6. «Рельсовое» программирование с сетями потоков данных	280
14.7. Регулировка обновлений о ходе выполнения операции	283
Приложение А. Поддержка унаследованных платформ	289
Поддержка аsync на старых платформах	290
Поддержка Dataflow на старых платформах.....	290
Поддержка System.Reactive на старых платформах.....	291
Приложение Б. Распознавание и интерпретация асинхронных паттернов	292
Асинхронный паттерн на основе Task (TAP)	293
Модель асинхронного программирования (APM)	294
Об авторе	301
Об обложке	302

Дополнительная информация

В рецепте 9.7 рассматриваются блокирующие стеки и мультимножества на случай, если вам потребуются сходные коммуникационные каналы без семантики «первым зашел, первым вышел».

В рецепте 9.8 рассматриваются очереди, имеющие асинхронный API вместо блокирующего.

В рецепте 9.12 рассматриваются очереди, имеющие как асинхронный, так и блокирующий API.

В рецепте 9.9 рассматриваются очереди с регулировкой количества элементов.

9.7. Блокирующие стеки и мультимножества

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одного потока в другой, но вы не хотите, чтобы этот канал использовал семантику FIFO.

Решение

Тип .NET `BlockingCollection<T>` по умолчанию работает как блокирующая очередь, но он также может работать как любая другая коллекция «производитель/потребитель». По сути это обертка для потокобезопасной коллекции, реализующей `IProducerConsumerCollection<T>`.

Таким образом, вы можете создать `BlockingCollection<T>` с семантикой LIFO или семантикой неупорядоченного мультимножества:

```
BlockingCollection<int> _blockingStack = new BlockingCollection<int>(
    new ConcurrentStack<int>());
BlockingCollection<int> _blockingBag = new BlockingCollection<int>(
    new ConcurrentBag<int>());
```


Важно учитывать, что с упорядочением элементов связаны некоторые условия гонки. Если вы позволите тому же коду-производителю обработать ранее любой код-потребитель, а затем выполните код-потребитель после кода-производителя, порядок элементов будет в точности таким же, как у стека:

```
// Код-производитель
_blockingStack.Add(7);
_blockingStack.Add(13);
_blockingStack.CompleteAdding();

// Код-потребитель
// Выводит "13", затем "7".
foreach (int item in _blockingStack.GetConsumingEnumerable())
    Trace.WriteLine(item);
```

Если код-производитель и код-потребитель выполняются в разных потоках (как это обычно бывает), потребитель всегда получает следующим тот элемент, который был добавлен последним. Например, производитель добавляет 7, потребитель получает 7, затем производитель добавляет 13, потребитель получает 13. Потребитель *не* ожидает вызова `CompleteAdding` перед тем, как вернуть первый элемент.

Пояснение

Все, чтобы было сказано о регулировке применительно к блокирующим очередям, также применимо к блокирующим стекам или мультимножествам. Если ваши производители работают быстрее потребителей и вы хотите ограничить использование памяти блокирующим стеком/очередью, используйте регулировку так, как показано в рецепте 9.9.

В этом рецепте для кода-потребителя используется `GetConsumingEnumerable` — самый распространенный сценарий. Также существует метод `Take`, который позволяет потребителю получить только один элемент (вместо потребления всех элементов).

Если вы хотите обращаться к совместно используемым стекам или мультимножествам асинхронно (например, если UI-поток должен действовать в режиме потребителя), обращайтесь к рецепту 9.11.

Дополнительная информация

В рецепте 9.6 рассматриваются блокирующие очереди, которые используются намного чаще блокирующих стеков или мультимножеств.

В рецепте 9.11 рассматриваются асинхронные стеки и мультимножества.

9.8. Асинхронные очереди

Задача

Требуется коммуникационный канал для передачи сообщений или данных из одной части кода в другую по принципу FIFO без блокирования потоков.

Например, один фрагмент кода может загружать данные, которые отправляются по каналу по мере загрузки; при этом UI-поток получает данные и выводит их.

Решение

Требуется очередь с асинхронным API. В базовом фреймворке .NET такого типа нет, но NuGet предоставляет пару возможных решений.

Во-первых, вы можете использовать Channels. Channels — современная библиотека для асинхронных коллекций «производитель/потребитель», уделяющая особое внимание высокому быстродействию в крупномасштабных сценариях. Производители обычно записывают элементы в канал вызовом `WriteAsync`, а когда они завершат производство элементов, один из них вызывает `Complete` для уведомления канала о том, что в дальнейшем элементов больше не будет:

```
Channel<int> queue = Channel.CreateUnbounded<int>();
```

```
// Код-производитель
ChannelWriter<int> writer = queue.Writer;
await writer.WriteAsync(7);
await writer.WriteAsync(13);
writer.Complete();
```

```
// Код-потребитель
// Выводит "7", затем "13".
ChannelReader<int> reader = queue.Reader;
await foreach (int value in reader.ReadAllAsync())
    Trace.WriteLine(value);
```

Более простой код-потребитель использует асинхронные потоки; дополнительную информацию см. в главе 3. На момент написания книги асинхронные потоки были доступны только на самых новых платформах .NET; старые платформы могут использовать следующий паттерн:

```
// Код-потребитель (старые платформы).
// Выводит "7", затем "13".
ChannelReader<int> reader = queue.Reader;
while (await reader.WaitToReadAsync())
    while (reader.TryRead(out int value))
        Trace.WriteLine(value);
```

Обратите внимание на двойной цикл `while` в коде-потребителе для старых платформ, это нормально. Метод `waitToReadAsync` будет асинхронно ожидать до того, как элемент станет доступным, или канал будет помечен как завершенный; при наличии элемента, доступного для чтения, возвращается `true`. Метод `TryRead` пытается прочитать элемент (немедленно и синхронно), возвращая `true`, если элемент был прочитан. Если `TryRead` вернет `false`, это может объясняться тем, что прямо сейчас доступного элемента нет, или же тем, что канал был помечен как завершенный, и элементов больше вообще не будет. Таким образом, когда `TryRead` возвращает `false`, происходит выход из внутреннего цикла `while`, а потребитель снова вызывает метод `WaitToReadAsync`, который вернет `false`, если канал был помечен как завершенный.

Другой вариант организации очереди «производитель/потребитель» — использование `BufferBlock<T>` из библиотеки TPL Dataflow. Тип `BufferBlock<T>` имеет много общего с каналом. Следующий пример показывает, как объявить `BufferBlock<T>`, как выглядит код-производитель и как выглядит код-потребитель:

```
var _asyncQueue = new BufferBlock<int>();

// Код-производитель.
await _asyncQueue.SendAsync(7);
await _asyncQueue.SendAsync(13);
```



```

_asyncQueue.Complete();

// Код-потребитель.
// Выводит "7", затем "13".
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.ReceiveAsync());

```

Код-потребитель использует метод `OutputAvailableAsync`, который на самом деле полезен только с одним потребителем. Если потребителей несколько, может случиться, что `OutputAvailableAsync` вернет `true` для нескольких потребителей, хотя элемент только один. Если очередь завершена, то `ReceiveAsync` выдаст исключение `InvalidOperationException`. Таким образом, с несколькими потребителями код будет выглядеть так:

```

while (true)
{
    int item;
    try
    {
        item = await _asyncQueue.ReceiveAsync();
    }
    catch (InvalidOperationException)
    {
        break;
    }
    Trace.WriteLine(item);
}

```

Также можно воспользоваться типом `AsyncProducerConsumerQueue<T>` из NuGet-библиотеки `Nito.AsyncEx`. Его API похож на API `BufferBlock<T>`, но не совпадает с ним полностью:

```

var _asyncQueue = new AsyncProducerConsumerQueue<int>();

// Код-производитель
await _asyncQueue.EnqueueAsync(7);
await _asyncQueue.EnqueueAsync(13);
_asyncQueue.CompleteAdding();
// Код-потребитель
// Выводит "7", затем "13".
while (await _asyncQueue.OutputAvailableAsync())
    Trace.WriteLine(await _asyncQueue.DequeueAsync());

```

В этом коде также используется метод `OutputAvailableAsync`, который обладает теми же недостатками, что и `BufferBlock<T>`. С несколькими потребителями код обычно выглядит примерно так:

```
while (true)
{
    int item;
    try
    {
        item = await _asyncQueue.DequeueAsync();
    }
    catch (InvalidOperationException)
    {
        break;
    }
    Trace.WriteLine(item);
}
```

Пояснение

Рекомендую использовать библиотеку `Channels` для асинхронных очередей «производитель/потребитель» там, где это возможно. Помимо регулировки поддерживаются несколько режимов выборки, а код тщательно оптимизирован. Однако, если логика вашего приложения может быть выражена в виде «конвейера», через который проходят данные, TPL `Dataflow` может быть более логичным кандидатом. Последний вариант — `AsyncProducerConsumerQueue<T>` — имеет смысл в том случае, если в вашем приложении уже используются другие типы из `AsyncEx`.



Библиотека `Channels` находится в пакете `System.Threading.Channels`, `BufferBlock<T>` — в пакете `System.Threading.Tasks.Dataflow`, а тип `AsyncProducerConsumerQueue<T>` — в пакете `Nito.AsyncEx`.

Дополнительная информация

В рецепте 9.6 рассматриваются очереди «производитель/потребитель» с блокирующей семантикой вместо асинхронной.