Оглавление

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 отработать 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.