В версию 4.0 среды .NET Framework добавлено новое пространство имен System.

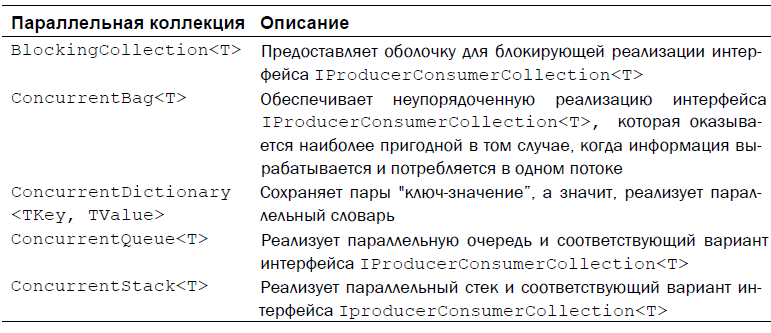
Collections.Concurrent. Оно содержит коллекции, которые являются потокобезопасными

и специально предназначены для параллельного программирования. Это

означает, что они могут безопасно использоваться в многопоточной программе, где

возможен одновременный доступ к коллекции со стороны двух или больше параллельно

исполняемых потоков. Ниже перечислены классы параллельных коллекций.



Как видите, в нескольких классах параллельных коллекций реализуется интерфейс

IProducerConsumerCollection. Этот интерфейс также определен в пространстве

имен System.Collections.Concurrent. Он служит в качестве расширения

интерфейсов IEnumerable, IEnumerable<T> и ICollection. Кроме того, в нем

определены методы TryAdd() и TryTake(), поддерживающие шаблон "поставщик-

потребитель". (Классический шаблон "поставщик-потребитель" отличается решением

двух задач. Первая задача производит элементы коллекции, а другая потребляет их.)

Метод TryAdd() пытается добавить элемент в коллекцию, а метод TryTake() — удалить

элемент из коллекции. Ниже приведены формы объявления обоих методов.

bool TryAdd(Т item)

bool TryTake(out T item)

Метод TryAdd() возвращает логическое значение true, если в коллекцию добавлен

элемент *item*. *А* метод TryTake() возвращает логическое значение true, если элемент

*item* удален из коллекции. Если метод TryAdd() выполнен успешно, то элемент i*tem*

будет содержать объект. (Кроме того, в интерфейсе IProducerConsumerCollection

указывается перегружаемый вариант метода СоруТо(), определяемого в интерфейсе

ICollection, а также метода ToArray(), копирующего коллекцию в массив.)

Параллельные коллекции зачастую применяются в комбинации с библиотекой

распараллеливания задач (TPL) или языком PLINQ. В силу особого характера этих коллекций

все их классы не будут рассматриваться далее подробно. Вместо этого на конкретных

примерах будет дан краткий обзор класса BlockingCollection<T>. Усвоив

основы построения класса BlockingCollection<T>, вы сможете без особого труда

разобраться и в остальных классах параллельных коллекций.

В классе BlockingCollection<T>, по существу, реализуется блокирующая очередь.

Это означает, что в такой очереди автоматически устанавливается ожидание любых

попыток вставить элемент в коллекцию, когда она заполнена, а также попыток

удалить элемент из коллекции, когда она пуста. Это идеальное решение для тех ситуаций,

которые связаны с применением шаблона "поставщик-потребитель". В классе

BlockingCollection<T> реализуются интерфейсы ICollection, IEnumerable,

IEnumerable<T>, а также IDisposable.

В классе BlockingCollection<T> определяются следующие конструкторы.

public BlockingCollection()

public BlockingCollection(int boundedCapacity)

public BlockingCollection(IProducerConsumerCollection<T> collection)

public BlockingCollection(IProducerConsumerCollection<T> collection,

int boundedCapacity)

В двух первых конструкторах в оболочку класса BlockingCollection<T> заключается

коллекция, являющаяся экземпляром объекта типа ConcurrentQueue<T>.

А в двух других конструкторах можно указать коллекцию, которая должна быть положена

в основу коллекции типа BlockingCollection<T>. Если указывается параметр

*boundedCapacity,* то он должен содержать максимальное количество объектов, которые

коллекция должна содержать перед тем, как она окажется заблокированной. Если

же параметр *boundedCapacity* не указан, то коллекция оказывается неограниченной.

Помимо методов TryAdd() и TryTake(), определяемых параллельно с теми,

что указываются в интерфейсе IProducerConsumerCollection<T>, в классе

BlockingCollection<T> определяется также ряд собственных методов. Ниже представлены

методы, которые будут использоваться в приведенных далее примерах.

public void Add(T item)

public T Take()

Когда метод Add() вызывается для неограниченной коллекции, он добавляет

элемент *item,* в коллекцию и затем возвращает управление вызывающей части программы.

А когда метод Add() вызывается для ограниченной коллекции, он блокирует

доступ к ней, если она заполнена. После того как из коллекции будет удален один

элемент или больше, указанный элемент *item* будет добавлен в коллекцию, и затем

произойдет возврат из данного метода. Метод Таkе() удаляет элемент из коллекции

и возвращает управление вызывающей части программы. (Имеются также варианты

обоих методов, принимающие в качестве параметра признак задачи как экземпляр

объекта типа CancellationToken.)

Применяя методы Add() и Таке(), можно реализовать простой шаблон

"поставщик-потребитель", как показано в приведенном ниже примере программы.

В этой программе создается поставщик, формирующий символы от А до Z, а также

потребитель, получающий эти символы. При этом создается коллекция типа

BlockingCollection<T>, ограниченная 4 элементами.

(***glava25\_19***)

class BlockingDemo

{

static BlockingCollection<char> bc;

//set symbols A to Z

static void Producer()

{

for(char ch = 'A'; ch <= 'Z'; ch++)

{

bc.Add(ch);

Console.WriteLine("Now symbol is: " + ch);

}

}

//use 26 symbols

static void Consumer()

{

for (int i = 0; i < 26; i++)

Console.WriteLine("Use symbol: " + bc.Take());

}

static void Main()

{

//use blocking collection with 4 elements

bc = new BlockingCollection<char>(4);

//create task producer consumer

Task Prod = new Task(Producer);

Task Con = new Task(Consumer);

//run tasks

Con.Start();

Prod.Start();

//wait both to be done

try

{

Task.WaitAll(Con, Prod);

}

catch(AggregateException exc)

{

Console.WriteLine(exc);

}

finally

{

Con.Dispose();

Prod.Dispose();

bc.Dispose();

}

}

}

Если запустить эту программу на выполнение, то на экране появится смешанный

результат, выводимый поставщиком и потребителем. Отчасти это объясняется тем,

что коллекция bc ограничена 4 элементами, а это означает, что в нее может быть добавлено

только четыре элемента, прежде чем ее придется сократить. В качестве эксперимента

попробуйте сделать коллекцию bc неограниченной и понаблюдайте за полученными

результатами. В некоторых средах выполнения это приведет к тому, что

все элементы коллекции будут сформированы до того, как начнется какое-либо их потребление.

Кроме того, попробуйте ограничить коллекцию одним элементом. В этом

случае одновременно может быть сформирован лишь один элемент.

Для работы с коллекцией типа BlockingCollection<T> может оказаться полезным

и метод CompleteAdding(). Ниже приведена форма его объявления.

public void CompleteAdding()

Вызов этого метода означает, что в коллекцию не будет больше добавлено ни одного

элемента. Это приводит к тому, что свойство IsAddingComplete принимает логическое

значение true. Если же коллекция пуста, то свойство IsCompleted принимает

логическое значение true, и в этом случае вызовы метода Таке() не блокируются.

Ниже приведены формы объявления свойств IsAddingComplete и IsCompleted.

public bool IsCompleted { get; }

public bool IsAddingComplete { get; }

Когда коллекция типа BlockingCollection<T> только начинает формироваться,

эти свойства содержат логическое значение false. А после вызова метода

CompleteAdding() они принимают логическое значение true.

Ниже приведен вариант предыдущего примера программы, измененный с целью

продемонстрировать применение метода CompleteAdding(), свойства IsCompleted

и метода TryTake().

(***glava25\_20***)

//set symbols A to Z

static void Producer()

{

for (char ch = 'A'; ch <= 'Z'; ch++)

{

bc.Add(ch);

Console.WriteLine("Now symbol is: " + ch);

}

bc.CompleteAdding();

}

//use 26 symbols

static void Consumer()

{

char ch;

while (!bc.IsCompleted)

{

if (bc.TryTake(out ch))

Console.WriteLine("Use symbol: " + bc.Take());

}

}

Этот вариант программы дает такой же результат, как и предыдущий. Главное

его отличие заключается в том, что теперь метод Producer() может производить

и поставлять сколько угодно элементов. С этой целью он просто вызывает метод

CompleteAdding(), когда завершает создание элементов. А метод Consumer() лишь

"потребляет" произведенные элементы до тех пор, пока свойство IsCompleted не

примет логическое значение true.

Несмотря на специфический до некоторой степени характер параллельных коллекций,

предназначенных в основном для параллельного программирования, у них,

тем не менее, имеется немало общего с обычными, непараллельными коллекциями,

описанными в предыдущих разделах. Если же вам приходится работать в среде

параллельного программирования, то для организации одновременного доступа к

данным из нескольких потоков вам, скорее всего, придется воспользоваться параллельными

коллекциями.